AFIT/GCS/ENG/93D-15

INTEGRATION AND ENHANCEMENT OF THE SABER WARGAME

THESIS
Karl Steven Mathias
Captain, USAF

AFIT/GCS/ENG/93D-15

Approved for public release; distribution unlimited

AFIT/GCS/ENG/93D-15

INTEGRATION AND ENHANCEMENT OF THE SABER WARGAME

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Karl Steven Mathias, B.S.

Captain, USAF

December, 1993

Preface

At the age of 12 I purchased my first wargame, *Squad Leader*, by the Avalon Hill Game Company. Little did I realize the impact $25 of game parts, boards, and tables would have on my life. It was from this game and the subsequent 50 games I purchased that I developed my interest in history, the military, and the simulation of combat. It should be no surprise to anyone then that I jumped at the opportunity to do research in the area of wargaming.

The first wargames, Wei Hai and Chaturanga, were played over 5,000 years ago. The precursors of Go and Chess, they were man's first attempts at military training through simulation (19). In the five millenia since their invention, we have moved from the board to the computer. In that transition we have attempted to remove the abstract nature of a "game" and introd[illegible] the thousands of variables that contribute to the fog of war.

Computer [illegible]ology and software engineering methodology have improved tremendously since wargames were first introduced on computers. However, with 5,000 years of history, wargames are slow to change and highly resistant to alteration. In my preparation for this research I was surprised to learn that many wargames, including the Air Force's primary training tool, are written in FORTRAN. Many lack even the standards of structured programming. Saber, which introduces object-oriented analysis, design, and programming is a major advance compared to the legacy code of many games.

I have taken special care in this thesis to look beyond just this one wargame. It is my hope that readers of this work will see applications of the concepts, design, and code for their own projects. My goal has been to make the spin-offs of Saber's research reap greater rewards than the satisfaction of completing this one project.

I would like to thank my thesis advisor, Major Mark Roth, for his guidance during this project. I am in debt to Major Scott Goehring at the Air Force Wargaming Center for his invaluable insight into military wargames. Most of all, I am grateful for the patience and support of my wife, Tracie, during the long nights and weekends this thesis was prepared.

Karl Steven Mathias

Table of Contents

List of Figures

Abstract

The Saber wargame is a theater-level air/land battle wargame intended for use by the Air Force Wargaming Center at Maxwell AFB, AL. Targeted for students in the Air Force Air Command and Staff College and the Air War College, Saber provides valuable simulation of how logistics, weather, intelligence, firepower, terrain, and unit posture influence combat.

This thesis documents how the separately developed components of Saber were integrated. A series of previous development efforts had resulted in the creation of two major subsystems: the simulation engine and the user interface. The communication mechanism between subsystems, a set of flat-files, had not been coordinated with the result that they could not exchange data. In addition, the user interface was missing vital data entry forms for entering combat unit orders, and was incomplete due to limitations of the Ada/X Window System bindings.

An abstract object-oriented database interface was developed for the system. The two components were then tied in to this interface. An object-oriented database, written in Ada, was selected and integrated with the database interface. The database interface proved to be flexible and allowed the integration of the two subsystems. It was discovered, however, that simply replacing a flat-file system with an OODBMS is not sufficient and can result in performance degradation.

An efficient technique for converting the user interface from the original Software Technology for Reliable Adaptable Systems (STARS) X Window System bindings to the newer Systems Engineering Research Corporation's bindings was developed. The user interface was converted and a land unit movement order entry form was implemented. It was found that much of the setup code in an X application can be broken into simple components resulting in significant code savings. This fact was utilized in developing visual components for the entry forms, allowing the land unit movement order entry form to be quickly constructed.

INTEGRATION AND ENHANCEMENT OF THE SABER WARGAME

I. Introduction

1.1 Overview

Short of war itself, the best training of field commanders occurs during large-scale field exercises. Unfortunately, these exercises present huge logistical problems, and are very expensive. At a theater of operations level, the logistic complexity makes the task nearly impossible. Computer simulation of combat at this scale gives the Air Force an inexpensive mechanism for supplying this type of training.

Saber is a theater-level air/land battle simulation designed for use by the Air Force Wargaming Center (AFWC). Its purpose is to augment the educational tools in use at the AFWC by supplying a wargame that runs entirely on a workstation without mainframe support. Additionally, Saber will be the first wargame in use at the AFWC that is written in Ada.

This thesis discusses how the components of Saber were integrated, optimized, and enhanced by an Ada Object-Oriented Database Management System (OODBMS) and a new set of commercial X Windows bindings. The research involved in this effort has resulted in the development of a highly flexible Ada interface that works with any OODBMS. It has also allowed the development of techniques for converting from old versions of X Windows bindings to new versions.

1.2 Background

A theater-level wargame, Saber is targeted for students attending the Air Force Air Command and Staff College (ACSC) and the Air War College (AWC). Intended as an educational tool, Saber simulates air and land combat at the theater level. Students using

this wargame will learn how logistics, weather, intelligence, firepower, terrain, and posture combine to influence the outcome of combat.

1.2.1 Purpose. Previous wargames used by ACSC and AWC, such as the Theater War Exercise (TWX), required numerous input forms and produced massive amounts of printed output. TWX had, in fact, been labeled the "Paper War" by students (12:4). Saber replaces this outdated system by supplying an easily understood graphic representation of the battle. The display of Saber closely approximates the ease with which board games (maps) physically represent combat. By making understanding of the combat situation more intuitive, the players will believe the game to better represent real combat.

1.2.2 Organization. Saber was designed in an object-oriented fashion so that different pieces could be reused for other wargames. This resulted in a separation into three main systems:

1. *User Interface.* The user interface uses the X Window System and OSF/Motif to display information. It contains two subsystems—the preprocessor and the postprocessor. The preprocessor allows the user to enter orders for the combat units. The postprocessor allows the user to see the results of the last segment of combat.
2. *Simulation.* A stand-alone program, the simulation engine executes the orders issued by the users and performs the associated combat. As it runs, the simulation maintains an event history. This history is passed back to the user interface for the postprocessor to display.
3. *Database.* The database system manages the basic information about the scenario needed by the user interface and simulation engine. This includes information such as terrain, weather, available combat units, weapons systems, supplies, etc.

1.2.3 Development History. The development of Saber has centered around three primary areas:

1. *Model.* Initial work on Saber began with the efforts of Marlin Ness. Ness sought to replace the land battle in TWX since air combat was having little effect other

than "to slow down the ground units." (15:2) To solve this, he developed a new land battle based upon the Lanchester equations and wrote some of the basic algorithms in Ada.

The land battle was now much better than the air battle in TWX. Mann (12) decided to replace the air model and link it with the land model. His final work integrated a stochastic air combat model with Ness's land battle. This resulted in an entirely new game, and it was given the name Saber.

2. *User Interface.* After Mann completed the model, Klabunde (10), Sherry (21), and Horton (9) took on the task of designing and implementing Saber in Ada. Klabunde, assigned the postprocessor subsystem, designed the user interface to run under X Windows and OSF/Motif. Working closely with the AFWC, he was able to reuse some of their interface components to construct the underlying basics for the system.

   Horton worked on the preprocessor and database system. He developed an initial set of user input forms that would be used to enter orders. Using the Oracle Relational Database Management System (RDBMS), he created a database layout to help with the entry and extraction of simulation data.

   Following Klabunde and Horton, Moore (14) performed research into moving Saber to a different set of X Window System bindings. He discovered that low-level mapping of X bindings made this type of conversion nearly impossible (14:36). Using the existing bindings, Moore was able to redesign the user input form developed by Horton and prepare a basic framework for using it.

3. *Simulation.* Sherry began work on the simulation engine by developing an object-oriented design for it. She attempted to develop the model's algorithms into Ada, but lack of experience in the language prevented her from getting far in this endeavor.

   Following Sherry, Douglass (6) re-evaluated Sherry's design and decided to re-engineer it using more modern object-oriented techniques. He repackaged the objects and instituted method access to attributes. Douglass implemented most of the Saber model, leaving only the air combat aspects unfinished.

1.3 The Problem

The simulation engine and user interface were still only partially complete. Through the development phases, the flat file formats used by each component had become inconsistent, and the components could not communicate. The challenge was to integrate these subsystems using a common database interface. This would allow a means of communication without creating data format dependencies.

The Software Technology for Adaptable, Reliable Systems Foundation (STARS) X Window System bindings had become obsolete since development began on Saber. They were no longer being maintained, and were in danger of becoming incompatible with newer versions of the X Window System libraries. In order to complete the user interface it would have to be converted to a better set of bindings.

1.4 Objectives

To create a functioning wargame, the following objectives had to be met:

1. Develop methods to convert X Windows applications written with older sets of bindings into the newer commercial sets.
2. Use these methods to convert the current Saber user interface to Motif 1.2.
3. Enhance the user interface to allow entry of missions for land units.
4. Design a database interface that can be used by both the simulation and the user interface to store information in a common database.
5. Alter the simulation and user interface to make use of the database interface

1.5 Assumptions

The following assumptions were made during the thesis effort:

1. The models of Ness and Mann are correct and do not require further research or modification.

2. Command, control, and communications are modeled by the player interaction in the game and not by the computer simulation.
3. Verification and validation of the combat model will be conducted by the Air Force Wargaming Center.
4. Naval operations are not modeled.
5. Air operations, while defined for this wargame, will be implemented after this thesis effort.
6. Only unclassified data and algorithms will be used.
7. The game should model combat in any scenario or theater of operations.
8. The graphical user interface code will be compiled using SunAda and the System Engineering Research Corporation (SERC) Ada/Motif bindings. The target system will be any SunAda supported workstation with an X Window System server.
9. The simulation will compile with any validated Ada compiler.
10. The user interface will act as the game controller for sequencing the execution of the preprocessor, simulation, and postprocessor.

1.6 Approach

The approach to completing the Saber project consisted of several steps:

1. *Re-engineer Database.* The flat file system was replaced and enhanced to allow for concurrent access and versioning. An abstracted database interface was developed to allow connection to any object-oriented database. The actual database selected was developed by Science Applications International Corporation (SAIC) under government contract. The design of the database schema was accomplished using established object-oriented database techniques.
2. *Convert Saber.* A technique called high-level mapping was used to convert the user interface to a new set of bindings developed by SERC. This brought the application up to Motif version 1.2 and solved several problems inherent in the old bindings.

3. *Enhance User Interface.* Generalized components were designed and implemented for creating user input forms. The land unit movement orders input form was created and tested using these new components.

1.7 Materials and Equipment

All work was conducted on Sun Microsystems's SPARC 2 computer systems. These workstations were already purchased and installed with SunAda and Ada/Motif. No additional equipment or software was required for this thesis.

1.8 Sequence of Presentation.

The thesis is divided into seven chapters. Chapter I, *Introduction*, has given a brief overview of the work. Chapter II, *Summary of Current Knowledge*, discusses background information that provides a basis for the thesis research. Chapter III, *Database Analysis and Design*, contains the object-oriented analysis of Saber's database requirements and the design that was constructed from that analysis. Chapter IV, *Database Implementation*, shows how the database interface was constructed. Chapter V, *User Interface Analysis and Design*, discusses the conversion technique used to move Saber to the SERC bindings. It also presents the analysis and design of the mission entry forms. Chapter VI, *User Interface Implementation*, shows how the components used to build forms were implemented in Ada. Chapter VII, *Analysis of Research*, analyzes the results of the research. Finally, Chapter VIII, *Summary and Recommendations*, summarizes the thesis and makes recommendations for future work.

II. Summary of Current Knowledge

2.1 Overview

The research in this thesis provides new techniques in two key areas: object-oriented databases and X Window System interfaces. In order to understand how object-oriented databases work, it is necessary to understand how object-oriented analysis and design occurs. A good understanding of the X Window System is needed to understand the problems that occur when interfacing it to Ada—a key challenge of this thesis effort.

This chapter provides background information from current sources on object-oriented analysis, object-oriented databases, and the X Window System. Information presented here is used as the foundation for the analysis and design chapters later in the thesis.

2.2 Object-Oriented Analysis, Design, and Implementation

Saber has been designed and implemented in an object-oriented manner. Using this technique, the developer analyzes the problem and divides it into entities that encompass specific state and behavior. These entities become the objects in the system. The designer further defines the relationships and interaction between the objects to determine how the system functions as a whole.

Rumbaugh, et al., describes an object-oriented development methodology called Object Modeling Technique (OMT) (20:5). Saber has been developed using OMT's four specific stages:

1. Analysis. During the analysis stage, the developer defines what the requirements for the system are. Objects are identified and their relationships are mapped using entity relationship diagrams. Implementation decisions are specifically avoided. Rumbaugh recommends defining three models during analysis: an object relationship model, a dynamic relationship model, and a functional model.
2. System Design. In this stage the system's architecture is determined. The application is broken up into subsystems and the interaction method (if any) between subsystems is defined. Resources (such as storage, processors, etc.) are allocated to

each subsystem. Control mechanisms (procedural, event-driven, etc.) are defined for each subsystem. The overall focus is on what needs to be done, independently of how it is to be done (20:198).

3. Object Design. During the object design phase, the object relationship model, dynamic model, and functional model are evaluated to determine the operations that must be implemented for each object class. Algorithms for these operations are designed. Structures for representing the relationships between objects are defined. Control mechanisms are created according to the system design.

4. Implementation. The final stage of OMT involves transforming the design into an executable system. This particular area is highly dependent on the language selected, as some languages, such as Ada 9X, C++, and Smalltalk directly support object-oriented programming while others, such as Ada 83, C, and FORTRAN do not.

2.3 Object-Oriented Databases

Object-oriented techniques give the developer a powerful tool for defining requirements and translating them into working code. These techniques are valuable not only in programming languages, but in defining database schemas for the next generation of database systems: object-oriented databases. This section discusses the composition of an object-oriented database, and reviews two recently developed Ada-language OODBMSs.

2.3.1 Definition. The precise definition of what features a database must contain to be called an object-oriented database is the source of some controversy. In this review, two Ada object-oriented databases are being considered. Two of the primary questions to be determined about the databases in question are: 1) What features do they support, and 2) Are these features sufficient to consider them OODBMSs?

In the "Object-Oriented Database System Manifesto" (3), the authors specify several areas that an OODBMS must support: complex objects, unique object identifiers, object encapsulation, support for types or classes, inheritance, late binding, computational completeness, extendibility, persistence, storage management, concurrency, and the ability to make ad hoc queries. The "Third-Generation Data Base System Manifesto" (25) also calls

for rules in the engine (triggers, constraints), collections of objects, updatable views of objects, and support for SQL.

Barry (4) compiled a checklist of attributes which can be used to classify and evaluate OODBMSs. He provides several main categories of features but does not attempt to define whether an OODBMS should have any of them. These categories are used to discuss the two Ada object-oriented databases considered for Saber.

1. *Object-Model.* What types of objects are supported? What type of support is there for object methods? Is polymorphism supported? How are objects encapsulated? How are types and classes supported? Is inheritance supported? How are relationships between objects defined?
2. *Schema.* How are schema's defined? What support exists for changing the schema? Can it be changed dynamically? What support exists for changing the methods of a class?
3. *Architecture.* What is the implementation of the database server? Does it support multiple clients? Where are methods executed, the server or the client? Does it support rules?
4. *Transaction Properties.* Are transactions atomic? What degree of consistency is maintained? Are long transactions supported? Are nested transactions supported?
5. *Persistence Transparency.* Are Ada data structures persistent, or do they have to be loaded into persistent database structures? Must the user explicitly tell the database to save an object or is this done automatically?
6. *Concurrency Control.* What form of concurrency control is used?

2.3.2 Classic-Ada with Persistence. Classic-Ada is a product of Software Productivity Solutions Incorporated (SPS). The basic Classic-Ada package is a preprocessor that extends the Ada language to allow class constructs. Objects may be dynamically instantiated, and a message passing mechanism alleviates the problems associated with late binding in Ada.

Classic-Ada with Persistence is an extension that allows class structures to be defined as persistent. Objects instantiated from persistent classes maintain their state between application executions. Two packages are provided with functions for managing the database and persistent objects. We summarize Classic-Ada with Persistence's OODBMS capabilities using Barry's categories:

1. *Object Model.* Classic-Ada augments the package structure by introducing classes into Ada. A class may contain instance variables and instance methods. Single inheritance is supported, and classes may override their parent class' methods (polymorphism).

   Objects are instantiations of classes. Dynamic instantiation is supported. Methods in objects are invoked by sending messages to them—the object determines at run time which of its methods should execute the message. According to SPS, this gives support to late binding, though it is unclear how an object could adjust its response to a given message.

   Objects reference each other with the use of a 32-bit object id. Though not defined as a limited private type, SPS recommends that it be treated as such and supplies relational operations with which to test it. The id remains valid across application executions so long as the target object is persistent. The id is not valid across executions for non-persistent types. There is no support for inverse relationships.

2. *Schema Development.* A schema in Classic-Ada is essentially the class structure as defined by the program. Once this structure has been fixed, it cannot be modified without losing all persistent data. According to the Classic-Ada manual:

   > "To ensure consistent views of persistent objects, Classic-Ada with Persistence requires that all applications accessing a common persistent object base are generated from a single state of the class library. This state is marked by the date and time when the Classic_Executive was generated." (22)

3. *Architecture.* Each application in Classic-Ada opens the database and is responsible for maintaining its integrity. Classic-Ada does not support concurrent access, so only

one application can open a database at any given time. Because of this no-server implementation, all methods are executed by each application.

4. *Transaction Properties.* The lack of a server greatly simplifies maintaining the integrity of the database. Since Classic-Ada keeps some of the database on disk and some in memory, all that is required is to close the database before application termination. Unfortunately, failure to close the database could leave it in an inconsistent state, and it is not clear from the documentation whether recovery is possible.

5. *Persistence Transparency.* Atkinson noted that database input and output code typically accounts for 30 percent of a system's code (2). It is important to determine whether an OODBMS can help provide significant savings through its persistence mechanisms.

   Classic-Ada offers total transparency to the programmer. After opening the database, the objects stored in it are available without any further system calls. There is no requirement to do an explicit save of an object, though SPS recommends closing and reopening the database periodically to ensure that objects buffered in memory get written to the disk.

6. *Concurrency Control.* As indicated previously, Classic-Ada does not support concurrent access to the database.

In summary, Classic-Ada provides extensions to the Ada language that support classes, inheritance, and dynamic instantiation of objects. It does not utilize a server and does not allow concurrent access to the database. Persistence of objects is maintained across executions, but the schema of the objects may not be altered without loss of data.

2.3.3 The SAIC OODBMS. The SAIC OODBMS was developed for the US Air Force to replace an existing COBOL-based system. As part of the contract, SAIC developed an object-oriented database system that was not tied to the application. This OODBMS, written in Ada, is owned by the government and distributable within its agencies.

The database is supplied as a set of Ada packages that compile into the database server. Some of these packages may need to be modified depending on the application. In general, however, the user will write a client program that interfaces with the server via a messaging system.

The SAIC OODBMS approaches the task of object management in an entirely different manner from that of Classic-Ada. Rather than extending the language, they have built data structures that allow classes to be defined and expanded. Classes are considered objects, and, according to the documentation, every object has a class—including the class object. Classes are defined by using a schema file which shows the inheritance and methods. Additionally, each class must have its methods placed into a package and be linked into the database server. SAIC supplies a core set of classes and methods to handle objects, classes, methods, collections, arrays, integers, floats, strings, etc.

The classes defined via the schema file are static and cannot be changed, modified, or derived from at run time. An alternative to modifying the database server each time an application needs a new class (as described above), is to define an Ada record to hold a database object's attributes. The application program then uses the SAIC OODBMS's BLOB class to store the entire record as a single entity in the database. Using this technique allows each application to define its own "classes" of data. The drawback to using this approach is that methods cannot be inherently associated with the class data.

The server consists of 40,000+ lines of code, and there is little documentation to explain its use. The information presented here was gained mostly by examining the database server and sample application code directly. Once again, we discuss the database system using Barry's categories:

1. *Object Model.* A class is defined in two pieces. First, a schema file indicates the hierarchy of classes in the system. Sections in this file define a class, indicate who its parent class is, and declare the methods available in the class. The contents of this file are placed into the database by an initializer. After this initialization process, the server may be invoked to run on the database.

The second part of a class definition is an Ada package called its resolver. The resolver package contains all the methods defined in the schema file. A special front-end resolves messages by directing them to the correct method. If a message does not resolve properly, it is passed on to the parent resolver. If the message does not invoke a method in the class hierarchy, an exception is raised by the database.

Object methods are defined in two ways: in the class resolver package, or by means of a multiple message method. In the first case, the method is an Ada subprogram compiled and linked into the server. It cannot be changed without recompiling and linking the entire system. Note that the resolver structure allows polymorphism since a method defined in a child class will be resolved before the same method in a parent class.

In the second case, a special database object called a MultipleMessageMethod is created. Essentially, this object can be dynamically "programmed" to issue a sequence of messages. This sequence of messages might involve getting a key from a dictionary object and then using it to return a BLOB object identifier from a sorted collection. In addition to the object messages, the MultipleMessageMethod can accept various control structures such as an if-then-else construct. This powerful device allows applications to dynamically write and modify their methods.

2. *Schema Development.* There are no tools available for changing the schema of a database without invalidating the data. In general, changing the application schema will require writing conversion routines to recover information. As discussed previously, stored methods may be changed at any time.

   Different applications may use the same database so long as they share a common schema. The class schema is built into the database when it is initialized, so all applications will have it available. Application-defined complex types, however, will have to be declared by each application at run-time. Some applications may use only subsets of this part of the schema.

3. *Architecture.* The system consists of one server and multiple clients. Each client communicates with the server by passing a shared memory segment that contains an

object message. The server accesses the segment and attempts to resolve the message by following the class hierarchy discussed above.

Methods are executed solely in the server. Class methods are actually compiled as subprograms in the server. MultipleMessageMethod objects execute their methods when an "execute" message is received. Since the MultipleMessageMethod method for "execute" is itself a compiled subprogram in the server, all the processing for this dynamic method is performed in the server.

4. *Transaction Properties.*

   Database transactions are initiated with a `Start_Transaction` call to the server and terminated with a `Commit` or `Rollback` call. Nested transactions are not allowed.

   The server allows only one transaction to execute at a time. In this manner it maintains serializability at the cost of performance. This limitation disallows long transactions, and the documentation encourages the use of short transactions for good multi-user performance.

   Only committed transactions are written to the database. Due to the non-concurrent nature of the database, a very simple recovery mechanism is necessary, since only the current transaction will have been lost during a crash.

5. *Persistence Transparency.* Persistence is not transparent as in Classic-Ada. Each class determines in its create mechanism (usually a method called `new`) whether instances will use persistent memory. This persistent memory is supplied by a memory page manager package.

   It becomes the responsibility of the programmer to copy Ada data structures into database objects. The object method used to accomplish this copying results in the object being saved. So while the programmer doesn't explicitly have to tell the database to save the object, they must still copy information into database objects.

6. *Concurrency Control.* As noted previously, the system allows only one transaction to execute at a time. Other transactions are blocked and must wait until the executing transaction completes.

In summary, the SAIC OODBMS provides static classes that may be changed or added to by creating new server packages. By using Ada record structures, applications can dynamically create complex data types and store them as objects. Methods may also be dynamically created and associated with objects via object identifiers. Only one database server is allowed, and it may only execute one transaction at a time. All methods are executed by the server.

2.4 The X Window System

In this part of the chapter we shift away from a background discussion of areas relating to integration, and study those which affect the makeup of the user interface. As noted in Chapter I, *Introduction*, the user interface was designed to execute under the X Window System environment. In this section the environment is defined and issues relating to developing Ada software in it are discussed.

2.4.1 Background. This section covers methods used in interfacing Ada to the X[1] graphic environment. Written in C and designed to interface with C programs, X provides an object-oriented model with a standardized set of graphic tools. Methods of interfacing to this system include binding to the C language calls or re-implementing X in Ada.

The Massachusetts Institute of Technology (MIT) developed X in partnership with Digital Equipment Corporation (DEC). Released in 1986, X supplies a flexible, object-oriented, graphic toolkit that is used to develop user interfaces in a standard manner. X has quickly caught on with the manufacturers of mid-range workstations. These users of X have formed into the X Consortium which now controls the development of the X Window System software (16).

MIT continues to hold a copyright on the system, but distributes the source free of charge. In addition, certain "toolkits" have emerged which ease the burden of programming in X. The standard toolkit, Xt Intrinsics, is also distributed free of charge.

[1]By convention, the single letter X is synonymous with "The X Window System"

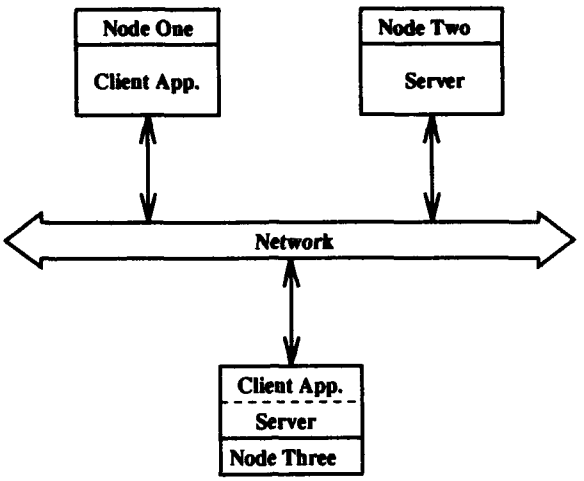

Figure 1. X Window System Client-Server Relationships

2.4.2 X Programming Model. The X Window System uses a client-server architecture that is designed to take advantage of networked systems. A server process runs on each display machine in a network (see Figure 1); it functions as the interface to that device's hardware. Client applications communicate with the server to have it display images and react when the user takes specific actions. Clients and servers may run on the same or different machines (16).

An X client application may utilize several layers of software, as shown in Figure 2. Each layer is an independent set of library calls, with each higher level having complete access to any of the lower layers. The layers are:

1. *X Library.* The lowest-level functionality of the X Window System is the X Library (Xlib). This library provides the very basic functions needed to render images on a display. Examples would be drawing lines, defining display colors, handling keystrokes, etc.
2. *X Toolkit Intrinsics.* Programming in Xlib can be a very labor intensive task due to the amount of event handling that takes place. Various toolkits have been developed that alleviate the burden of programming in Xlib, the most popular being the X Toolkit Intrinsics (Xt). Applications using Xt are given general facilities that allow streamlined event handling and the ability to create reusable display objects (like windows or buttons) called widgets.

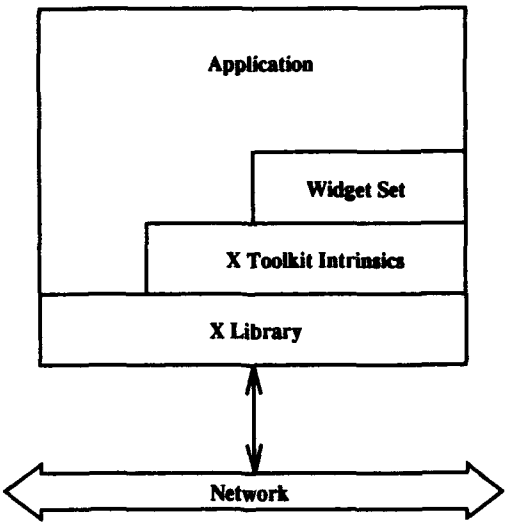

Figure 2. X Window System Libraries

3. *Widget Sets.* The X charter promotes "mechanism, not policy" (17). Thus, unlike other windowing systems such as Microsoft Windows, X does not force developers to have a mandated style to their application. Commercial widget sets such as the Open Software Foundation's (OSF) Motif widget set fill in this gap. Layered on top of Xt, the OSF Motif library is used according to a style guide, and provides a consistent look and feel across applications written with it (8).

2.4.3 Ada Language Bindings to X. The X, Xt Intrinsic and Motif libraries are written in C with no attempt at making them language independent. This presents a serious problem for DOD agencies who must write applications in Ada. The solution to this problem involves writing interface routines that allow Ada procedures to make the X procedure calls indirectly. These interface routines are called bindings.

Several sets of commercial and public domain bindings exist for the Ada developer:

- Software Technology for Adaptable Reliable Systems (STARS) Foundation SAIC and Boeing Sets (Binding)
- Ada/Motif by Systems Engineering Research Corporation (Binding)
- TeleWindows by TeleSoft (Binding/Builder)
- Builder Xcessory by Integrated Computer Solutions Inc. (Builder)

- ezX by Sunrise Software Int. (Builder)
- TAE+ by NASA Goddard Space Flight Center (Builder)

Some of these bindings sets are only partial implementations of X, Xt and Motif. Several just build interfaces by generating X calls and do not supply a programmer accessible set of bindings. This review concentrates on the two binding sets used by Saber, STARS and Ada/Motif.

2.4.4 Thick and Thin Bindings. Simple C language calls can easily be represented in the binding set by a corresponding Ada call. Complex C calls, however, require much more work to convert and return values properly. The level to which a binding processes information before passing it on is described in terms of thickness (7).

Thin bindings characterize themselves by a one-to-one mapping of Ada and C functions. Use of Ada specific features appears only when a C-specific construct may not be available. Within the bindings, minimal amounts of conversion and processing occur before making a call.

Thick bindings characterize themselves by heavy use of Ada constructs and language features, generalization of the mapping to C calls, and the addition of utility procedures/functions. A thick binding seeks to insulate the Ada programmer from the idiosyncrasies of C calls while giving them the freedom to utilize Ada features to their best advantage. Within the bindings, extensive processing and conversion will be required to build the structures needed by the corresponding C function(s).

The debate on the relative merits of thick and thin bindings is on-going (7). Thick bindings, which can be very close to implementations, offer the benefits of Ada's strong typing and are not so closely bound to C constructs. Thin bindings, on the other hand, are easier to build and maintain, an important consideration since the X Consortium will regularly release updates to X.

There seems to be some general agreement that an Ada interface specification needs to be adopted by the X Consortium. The X Consortium, however, shows little interest in creating any such standard (11). It appears that with X being updated every two to three

years, easily maintained or commercially supported thin bindings are the best choice for development.

2.4.5 STARS Binding Sets. Three major sets of bindings were developed under the STARS Foundation contracts: Science Applications International Corp (SAIC) Xlib bindings, Boeing's Xt Intrinsics/OSF Motif bindings, and the Unisys Ada/Xt software. The Unisys software, however, is not a true binding set as the actual Xlib and Xt Intrinsics libraries have been re-coded in Ada. This section concentrates on the SAIC and Boeing bindings.

2.4.5.1 SAIC Xlib Bindings. The SAIC bindings cover most of the Xlib functions, and are characterized as thin bindings. It should be noted, however, that they do make use of Ada specific features (11). As they were the first set to be developed for Ada, many projects using X employed them. Placed in the public domain, the SAIC bindings may be used free of charge. For those projects requiring only basic interfaces to X, these bindings may be adequate for the task.

There are, however, problems with the bindings. There is no documentation for the bindings, so to determine functionality, the programmer must read the source code (10). Additionally, the SAIC bindings do not precisely follow the X naming conventions. Some structures, functions and procedures have no counterparts in X, and the programmer must read the source to determine how to use them.

No on-going support is provided or planned for the SAIC bindings. As X continues to change, it will be up to the end-user to modify and extend their version of the bindings. This is a problem since the interface has bugs that must be fixed by "each and every user" that develops with them (7, 10, 14).

Last, no toolkits or widget sets support the internal structures of the SAIC bindings. Programmers must write interface and conversion routines to utilize toolkits like Xt Intrinsics and widget sets like OSF Motif.

2.4.5.2 Boeing Xt Intrinsics, OSF Motif Bindings. Like the SAIC bindings, the Boeing set is public domain. It implements a large portion of the OSF Motif func-

tionality, and has a complement of supporting bindings to Xt and Xlib. These bindings may be used stand-alone for simple Motif applications, or may be combined with the SAIC bindings for more functionality. The Boeing bindings are also public domain and may be used free of charge. While not under on-going support, the bindings have been found to be relatively bug-free (10:36).

There is no documentation for these bindings either, so usage is learned by reading comments from the source code. Only a limited subset of Xlib is supported, so any application that requires access to primitives will need custom bindings written or an interface to a full set of Xlib bindings (such as SAIC).

As with SAIC, the lack of on-going support is a problem. As OSF releases new versions of Motif, the Boeing bindings become more and more obsolete.

2.4.6 SERC Binding Set. SERC has released a commercial binding, Ada/Motif, which provides a complete Ada binding to X. This includes Xlib, Xt Intrinsics, the Athena widget set (Xaw), and OSF/Motif. SERC based their work on the STARS binding, debugging and expanding as needed (24:1). The following sections discuss how the bindings are organized, what documentation is available, how complete they are, and the advantages/disadvantages of using them.

2.4.6.1 Organization. While SERC indicates they based their work on STARS, the final product is actually quite different. This is evident in the nesting which Ada/Motif uses in its packages. The Xlib package, for example, is broken down into Atoms, Fonts, Colors, Graphic_Output, Cursors, Cut_And_Paste, Regions, Keyboard, Events, Window_Manager, and Resource_Manager sub-packages. The Xt and Xm packages break out similarly. In STARS, all of the functions for Xlib were contained in a single package called X_WINDOWS. This was also true for the STARS Xt and Xm packages.

Ada/Motif generates a nearly one-to-one mapping of X types, functions, and procedures to Ada types and calls. It is only in cases where Ada cannot support specific C constructs that special types or procedures are created. This usually occurs when a C function modifies its input parameters; in this case Ada must use a procedure to make

the call. Because of this close mapping, the SERC bindings can be characterized as thin bindings.

2.4.6.2 Documentation. The SERC documentation comes in two volumes, a user's guide and a print-out of the specification. The first volume describes how the bindings are organized, naming conventions, calling conventions, and discusses issues of interfacing Ada to C. Several annotated examples provide information on how to develop and debug an Ada/Motif application. A section also explains how to develop new widgets under Ada/Motif.

Volume one contains an index that allows the user to locate packages, procedures, and functions in volume two. This is invaluable as the nesting of packages often makes it difficult to determine in which package a function or procedure resides. Unfortunately, the index does not list X identifiers or types, and it is sometimes difficult to figure out which sub-package they are in. For example, Xt_String and Xt_String_List are in package Xt, but Widget is in package Xt and Xt_Widget_List is in package Xt.Ancillary_Types.

2.4.6.3 Completeness. SERC's claim of having a complete binding to Xlib, Xt, Xaw, and Xm appears to be correct at least for Xlib and Xt. Cross-referencing the Xlib and Xt packages with *The X Window System in a Nutshell* (1) (a complete reference to Xlib and Xt calls and types) yielded no deficiencies in Ada/Motif. The Xaw and Xm packages were not as extensively checked, but those functions and types examined matched correctly.

There were cases where Ada simply would not support a particular type of X call, specifically the variable argument calls. SERC has worked around this by supplying a function or procedure with the same name, but instead of passing variable arguments, an argument list is passed. This makes the variable function have an equivalent parameter list to the non-variable functions (24:5).

2.4.6.4 Naming Conventions. Ada/Motif uses a fairly simple naming convention scheme: the first letter of any identifier is capitalized and there are underscores

between words. Where conflicts occur with reserved words (such as New) the name is changed to be similar to the reserved word (X_New) (24:7).

2.4.6.5 Advantages. Ada/Motif has documentation that is adequate for developing applications using the bindings, certainly better than the total lack of documentation for STARS. The bindings provide complete X, Xt, Xaw, and OSF Motif functionality to the Ada programmer.

As a commercial product, Ada/Motif is being supported by SERC and is now on its second release (version 1.1). Because of this support, it appears to be gaining acceptance as other companies are using it as the underlying bindings to their interface builders (e.g., Integrated Computer Solution's Ada Xcessory).

2.4.6.6 Disadvantages. Any site developing with Ada/Motif must purchase a license since the product is not public domain. Initial releases required a copy of the license for every machine that used the bindings (14), but the latest version only requires a license for the development machine.

2.5 Summary

This chapter has discussed three areas of primary interest for integrating and enhancing Saber. Object-oriented techniques give the developer powerful tools for defining a project's requirements and translating them into a working system. Object-oriented database have their roots in this method of modeling. Two OODBMSs written in Ada were discussed: Classic-Ada with Persistence and the SAIC OODBMS developed for the government. The next chapter will show how the Saber database interface was designed using object-oriented techniques. The X Window System provides a flexible environment for creating visual applications. Though written in the C language, binding libraries are available to the Ada developer. Two bindings, Ada/Motif and STARS were discussed and compared.

III. Database Analysis and Design

3.1 Overview

This chapter describes how the requirements for Saber's abstract database interface were analyzed and how the final design was formulated. Saber uses this interface to interact with an OODBMS. The interface was carefully designed to hide the implementation detail of the underlying database. The interface itself was designed not only to allow use with Saber, but to have the flexibility to work with a wide class of applications needing general access to an OODBMS.

3.2 Analysis of Database Requirements

Both the Saber user interface and simulation engine originally read data from flat-files at the start of execution, and wrote this data back at completion. This was accomplished by having each object in the simulation receive an initialize message. The object would then open a file whose name had been hard-coded into the object. Using various algorithms, including several of quadratic complexity ($O(n^2)$), the data was loaded into internal structures.

3.2.1 Problems with the Original Database System. This method had many disadvantages. First, it required tight data coupling between the user interface and the simulation engine on the precise format of the flat files. In many cases, the user interface had to read and write data it wasn't going to use. Second, the up-front load required large amounts of memory since all the data was loaded at one time. Third, the $O(n^2)$ performance found in several of the routines caused read times to become progressively worse as the size of the scenario increased.

The solution to these problems was to introduce an underlying database that could support both the user interface and the simulation requirements without forcing them to consider physical data format requirements. The database had to allow the system to dynamically access data items so as to avoid the up-front load requirement. Finally, the database needed sufficient flexibility to establish object relationships such that simple access algorithms could quickly select related objects.

3.2.2 Relational versus Object-Oriented Database Management Systems. A relational database management system (RDBMS) could meet all of these requirements. An interface could be constructed that supplied each subsystem a format-independent view of the database. With carefully constructed structured query language (SQL) statements, dynamic access to any object stored in the database could be achieved. Structuring of the database could also allow traversal of item relationships without invoking inefficient search algorithms.

An object-oriented database, however, appeared more suitable to the job. Like the RDBMS, it could supply a format-independent view of the database. Unlike the RDBMS, however, direct access to any object could be achieved by referencing its object identifier, circumventing the need to parse an SQL statement. Since the object-relationships of each subsystem would be reflected in the database, traversal of object relationships could be accomplished without special database algorithms. If a persistence scheme were used, there would be no separate database code at all.

3.2.3 Identification of Simulation Objects. In order to correctly analyze the requirements for the database, we elected to apply OMT (20) to this problem. This decision was made since the underlying object diagrams for the OODBMS would be very similar to the object diagrams developed for the application. By analyzing the existing object diagrams of Saber, it would be possible to map these into database objects.

Douglass had previously identified the object classes in his thesis on Saber (6). As he had worked only on the simulation, there was concern that these might not match up properly with those used in the user interface. By carefully comparing the simulation code, user interface code, and Douglass' results, a set of database objects was identified. These objects are listed in Appendix B. Other than some semantic changes in attribute names, no significant differences between the user interface and simulation engine were found.

3.2.4 Identification of Object Relationships. Having identified the objects used in the system, it was necessary to determine how they were related. Once again, Douglass had

conducted research into this area and it was a simple manner of comparing the code and his diagrams. The diagrams resulting from this comparison are contained in Appendix A.

In this case, it was much more difficult to ascertain whether the relationships between objects matched. The original Saber wargame had been designed to read the flat files as if they were tuples in a relational database. As a result, internal structures were designed to allow for "joins" to be performed against different files. Douglass removed these internal structures when he re-designed the simulation engine in an object-oriented manner. The user interface, however, still contained these constructs. Thus, there was significant difference in how the data was internally stored between the two subsystems.

Looking past these types of constructs, however, revealed that most of the diagrams matched. The main exception dealt in the area of terrain, where the user interface has a "neighbor-link" object that indicates whether hexes are neighboring. The simulation engine determines this mathematically, and thus has no need for the object class.

3.2.5 Identification of Object Methods. It was assumed that object functionality would be supplied outside of the database. Only database management methods would be defined for each database object. How these database methods were determined is discussed in the next section.

3.3 General Design of Database Interface

3.3.1 Design Objectives. The primary theme of the work on Saber was to develop reusable components that would be able to work with any wargame-type simulation. Along these lines, the database interface design had several objectives intended to enhance its stability and re-usability:

1. *Be Database Independent.* As noted previously, both object-oriented and relational databases can support Saber's requirements. The interface developed had to sufficiently abstract the database management system implementation such that any database could be utilized. Obviously a RDBMS would require more work to im-

plement, but it was unknown whether an Ada OODBMS, or bindings to a C++ OODBMS would be available.

2. *Be Application Independent.* The requirement that the interface support both the simulation engine and the user interface mandated that it not be tied to a specific subsystem. Extending this, it was logical that the interface not necessarily be tied to the Saber system at all.

3. *Support Object Management.* Support of a wide range of applications required that the database implement a number of the operations available in most OODBMS. This included the ability to create, read, modify, and delete objects as well as form them into sets and query from these sets. Each object needed the ability to establish the relationships, one-to-many, one-to-one, or many-to-many, that are shown in object class diagrams.

4. *Support Persistent Data Types.* Applications that desire complete database transparency require persistent data types. The interface had to be able to support, either directly or indirectly, data types that could be made persistent.

5. *Support Transaction Control.* Transaction control is key to maintaining the integrity of the database. The interface had to implement basic forms of transaction control that could protect the atomicity of a database operation. This control had to be generic such that the application did not have to know the underlying database implementation.

3.3.2 Database Abstraction Layers. The design goals were achieved by modeling the system as a series of layers, an approach suggested by Rumbaugh (20). Each layer is its own virtual machine, with no knowledge of the lower layer's implementation or any upper layer's existence. Each layer, as shown in Figure 3, builds upon the immediate lower layer.

This architecture is similar to that used by McKay and Pederson when they constructed an Ada interface to an RDBMS (13). They found that this approach reduces the amount of repetitive code written for database access, simplifies programmer training requirements, and standardizes program interfaces.

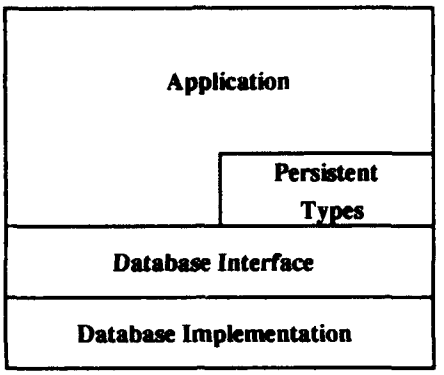

Figure 3. Database Abstraction Layers

3.3.2.1 Database Management System Layer. The bottom layer represents the actual database system implementation. The only assumptions made about this layer is that functional entry points are provided that allow operations to be performed on database entities. The entities could be database objects, persistent data types, or tuples in a relation.

3.3.2.2 Database Interface Layer. The database interface is a series of function/procedure calls and data structures whose implementations map into the DBMS layer's entry points. Internally, the interface layer is shown as consisting of an abstract database object class. The user declares sub-classes that inherit the behavior of this class and modify its virtual functions and procedures. A sample class diagram for an application is shown in Figure 4.

3.3.2.3 Persistence Layer. Persistence is based upon the database interface, and not directly upon the database. Implementing it at this level avoids portability problems between different databases, but is more costly in terms of performance. A properly implemented persistence scheme, however, with memory caching features should minimize this effect. The only persistent data types that could be used with Saber are the array and doubly-linked list. Implementations of these using the database interface are discussed in the next chapter.

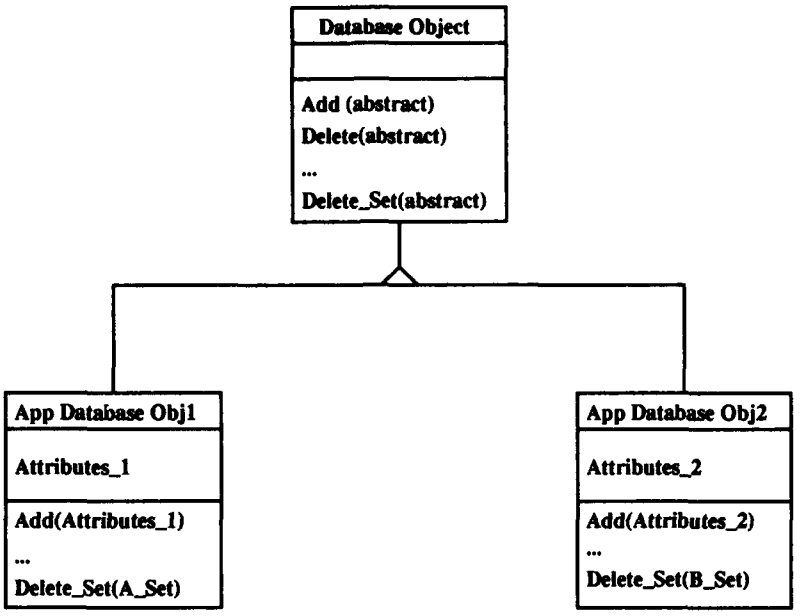

Figure 4. Object Class Diagram for Database Interface

3.3.2.4 Application Layer. The application may utilize persistent data types, the database interface, or both. In general, however, the application will create object classes from the database interface's database object.

3.3.3 Application/Interface Operations. Examination of the Saber simulation code revealed that the basic functions needed for an object were Create, Get, Put, and Delete. Additionally, operations to iterate over a class of objects were required. A sample of simulation engine specification code is shown in Figure 5. Note that there is no function to create objects, as this was assumed to have been done previously by a database builder. The functions that return object attributes (such as Get_Land_Units) dictate the existence of a Get function. The functions that alter attributes (such as Move_In_Grid) dictate the existence of a Put function. The ability to delete some objects (such as the Delete_A_Land_Unit procedure) requires that Delete be supplied. Iteration over a class of objects is supplied by the Set_First_Land_Unit procedure and the Next_Land_Unit procedure. This mandates similar functions in the interface. To iterate over the class, the First and Next operations are used. Sets representing object relationships are iterated over by using the

```
...
procedure Get_Land_Units;
procedure Set_First_Land_Unit;
procedure Next_Land_Unit(Land_Unit_No : out Natural;
                         Another_Land_Unit : out Boolean);
procedure Delete_A_Land_Unit (Land_Unit_No : Natural);
procedure Move_In_Grid (Land_Unit_No : Natural);
procedure Determine_Firepowers;
...
```

Figure 5. Saber Simulation Specification Code Fragment

First_XXXXX and Next_XXXXX functions where XXXXX is the name of the relationship set.

Saber had been designed in an object-oriented manner, but was not made to use object-oriented databases. Certain functionality had to be added so that the database could be opened and closed, and transactions protected. For this reason, Open_Database, Close_Database, Start_Transaction, Commit_Transaction, Rollback_Transaction are in the database interface. Additionally, extension of Saber might require that set operations be given better support, so the ability to form sets through class-level queries (Extract_Set) was added. Support operations to iterate over these sets and delete them when finished were also introduced. The complete set of operations is shown in Figure 6.

It is assumed that all object classes will be known at compile-time. The application must register all of its classes with the database interface so that the proper operations can be created for them. Since not all languages can support dynamic function binding, Ada in particular, the registration must occur at compile time. Given that the need to create classes "on the fly" should be rare, this is not a great limitation.

3.3.4 Interface/OODBMS Operations. The implementation of the operations present in the interface is highly dependent on the capabilities of the database. Because of this, the interface operations are generalized and simplified. Some databases, such as SAIC and ObjectStore directly support the concept of a unique object-identifier for each object. For other databases, such as Oracle's RDBMS, the object-identifiers might have to be generated by the interface implementation code.

```
-- Database Operations
Open_Database(in Database_Name)
Close_Database
Copy_Database(in Old_Database_Name, in New_Database_Name)
Start_Transaction
Commit_Transaction
Abort_Transaction

-- Database Object Class Operations
Create(in User_Data [, in Key]) return Object_Id
Get_ObjID(in Key) return Object_Id
Get(in Object_Id) return User_Data
Put(in Object_Id, in User_Data)
Delete(in Object_Id)

First_In_Set(out Object_Id, out Status)
Next_In_Set(in Object_Id, out Object_Id, out Status)
Add_To_Set(in Object_Id)
Remove_From_Set(in Object_Id)

Add_Object(in Object_Id)
Get_Object(in Object_Id)

Extract_Set(in Expression, out Object_Id)
Delete_Set(in Object_Id)
```

Figure 6. Database and Database Object Class Operations

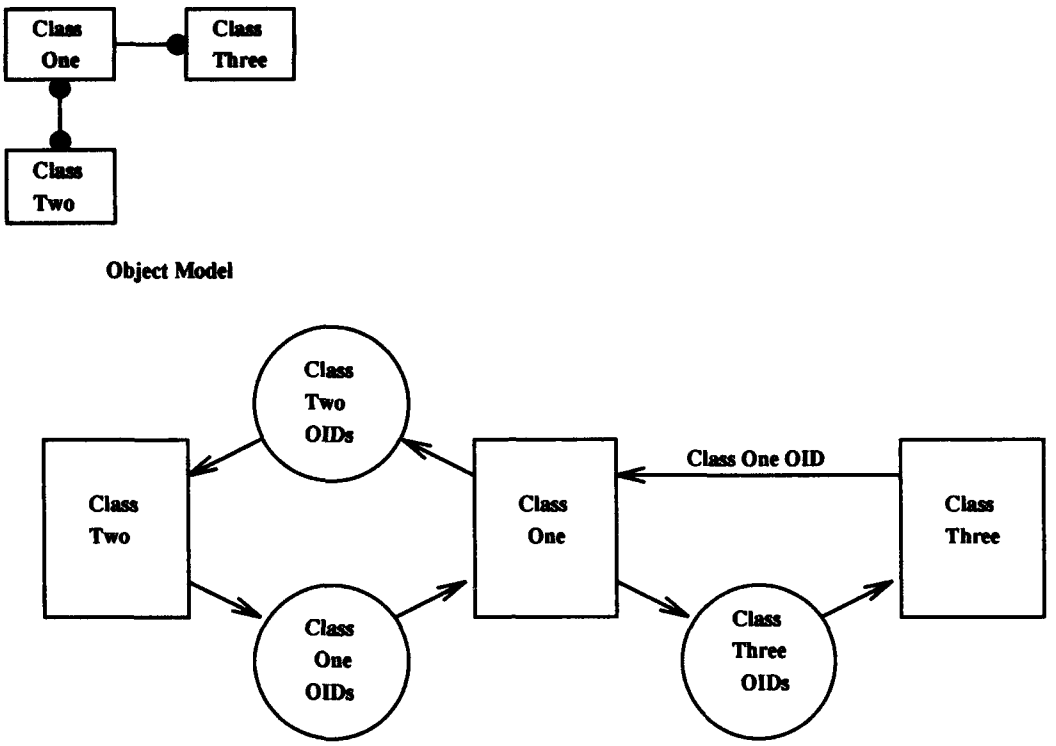

Figure 7. Database Storage Model

Complicating the issue is the creation of relationships for each object. The interface does this by creating sets that hold the object-identifiers that the object is related to in some manner (see Figure 7). Most OODBMSs directly support the concept of dynamically forming sets and modifying them. An RDBMS would have a much tougher job—it would have to create a master relation for each set possible and then be able to select a specific object's relationships from that relation.

Figure 7 shows how the database interface maintains relationships. In the upper left corner a simple Rumbaugh-style diagram shows the class relationships. The larger diagram shows how the interface would model this diagram. Rectangles indicate classes, circles sets of object identifiers, and arrows a single object identifier with the object it points to. Object class two would have a set of object class one identifiers. Selecting a specific identifier from that set would allow access to an instance of object class one. This could then be used to access the set of object class three identifiers associated with object one. A specific identifier could then be selected to get an instance from object class three.

3.3.5 Persistent Data Type Operations. There are no explicit operations on persistent data types, rather, this section outlines guidelines to follow when developing persistent types based on the interface. We recommend the following:

1. *Initialization.* In order to maintain the transparency of the persistence, the initialization (including loading of any data) should occur when the database is opened. This requires that the interface be able to register a persistent data type so that it can be properly set up when the the database is opened. The expected method of performing this is to make the "type" an object class that the interface recognizes.
2. *Finalization.* Finalization of an object occurs just prior to destruction. It is typically invoked to deallocate system resources that an object may be utilizing. Just as initialization occurs when the database opens, finalization should occur prior to a close.
3. *Declaration.* Persistent data types should be valid types in the implemented language. Special keywords that require a preprocessor are not desirable as they require development of additional database support software, i.e., the preprocessor itself. With all persistent types the user should be allowed to specify whether items in the structure should be pre-loaded and locked into memory, locked into memory as they are read, or never locked in memory. Databases which do not support locking of specific database items could just implement this as a null function.
4. *Modification.* Changes to any value should be reflected into the database. The structure must be implemented as write-through since other programs may also be accessing the persistent structure in the database simultaneously. The modification algorithm should handle transaction control internally, the granularity of the control should be for each modification made.

3.4 Saber Design Changes

3.4.1 Design Objectives. As discussed previously, the components of Saber were not integrated by a common database. The flat file format each subsystem expected was no longer common between the components. Additionally, since the flat files only simulated

an RDBMS, the performance of the loading algorithms was expected to seriously degrade as the scenarios grew. To attack these problems, two specific design objectives had to be met:

1. Common Data Source. The specific design of the database interface had to create an information manager for the user interface and simulation engine. The information manager would act as a control, storing and issuing data objects to the subsystems while hiding the format and implementation mechanisms.
2. Efficient Database Access. Elimination of the $O(n^2)$ algorithms in the load routines of the subsystems was a high priority. Instead of performing joins, the data objects would be loaded by traversing the object class diagrams for Saber. Since each object knows how to lookup any object it is related to, this eliminates the need for a join operation.

3.4.2 Approaches. There were several methods by which the subsystems of Saber could be altered to accept the new database. The following sections discuss these options and their advantages/disadvantages.

3.4.2.1 Persistent Data Types. A common structure found in the simulation engine was the doubly-linked list. This was introduced by Douglass when he rewrote the simulation to be more object-oriented. It eliminated the fixed arrays that limited the number of objects that could be maintained in memory. It is used to track all objects except ground hexes and weather forecasts, where the array is a more natural form to access these objects.

Replacement of the doubly-linked list is a simple matter. A persistent version of the doubly linked list is constructed by altering the implementation of its methods. The methods are changed to read and write database objects rather than in-memory pointer structures. A declaration switch activates memory locking for a structure in the database cache. This implementation is discussed further in the next chapter.

The advantage of using this persistent data type would be the complete transparency of the change within the simulation engine. Other than opening and closing the database,

```
-- Old Code
Old_Hex := Ground_Grid(X, Y);
Ground_Grid(X, Y) := New_Hex;

-- Persistent Array Code
Old_Hex := Ground_Grid.Retrieve(X, Y);
Ground_Grid.Assign(X, Y, New_Hex);
```

Figure 8. Impact of Changing to Persistent Arrays

this would entirely eliminate any database code from the application, removing a significant maintenance element.

Unfortunately, the two largest structures in Saber are arrays. Implementation of an array type to handle the hexagon ground grid and the weather forecasts would require changes to the application since Ada83 cannot overload the assignment operator or the array operator. A package similar to the doubly linked list would have to be constructed that handled persistent arrays. This would include functions to assign and retrieve values. The functional interface would result in changing all of the application's code that read a value from the array or changed a value in it. Figure 8 shows an example of how the changes would impact code.

An object class diagram for a persistent array is shown in Figure 9. An object class called Array_Object is a descendent of the base class, Database_Object. It implements the two abstract methods Create and Put that were inherited from Database_Object. The attribute App_Data stores an individual piece of data in the array. The Persistent_Array class knows the minimum and maximum bound of the array and is associated with zero or more Array_Objects. The Assign and Retrieve methods invoke the appropriate Array_Object's method to Get, Put, or Create its App_Data.

A state diagram of how a persistent array would work is shown in Figure 10. The persistent array class sits idle until an assignment or retrieval is attempted. When an assignment occurs, the new data is copied into an Array_Object. When this is complete, the object is stored in the database, this may occur via the Create or Put method. If

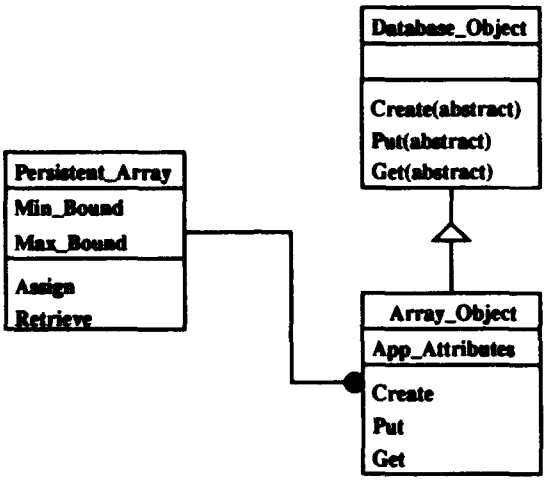

Figure 9. Object Model for Persistent Arrays

an error occurs, an error handler is entered. If a retrieval is attempted, the database is consulted to find the correct Array_Object. If it is found, the Get method is activated to copy the data into an application-supplied data structure. Should an error occur during OODBMS lookup, an error handler is invoked.

3.4.2.2 Persistent Objects. When Douglass encapsulated all the objects in the simulation, he created attribute access methods and iterators for all of the classes. The methods and iterators operate on memory structures encapsulated by the object. Methods to read and write the data to the database were included with each object so that its state could be maintained.

We can make object classes in Saber persistent by having them inherit from the database object class. As shown in Figure 11, we inherit the database operations from the parent. The simulation methods and the attribute access/modification methods are modified to operate on the database using the inherited database methods. This eliminates the need for in-memory structures, pre-simulation loading, and post-simulation writing of data.

Figure 12 is the state diagram for a persistent object. This diagram shows how the object would behave when an attribute modifying method is invoked.

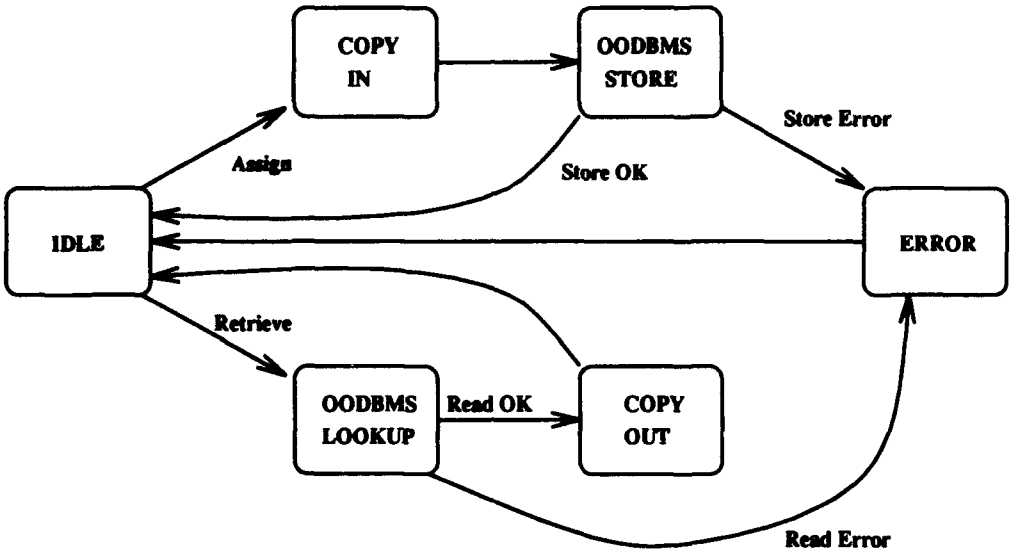

Figure 10. State Diagram for Persistent Arrays

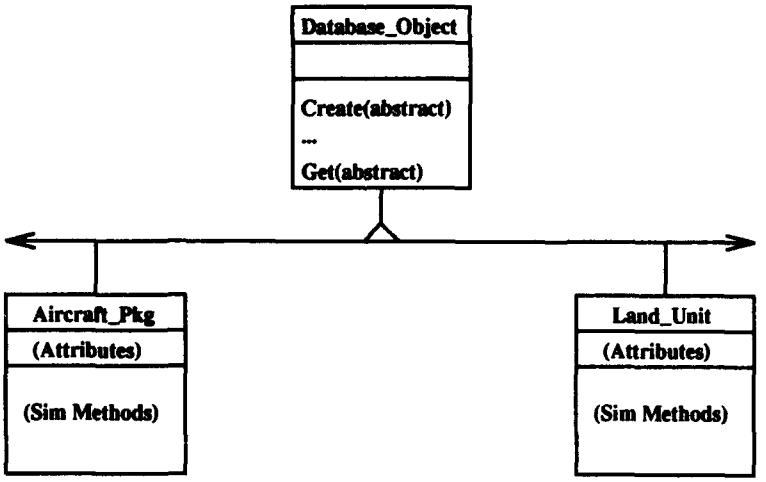

Figure 11. Object Model for Persistent Objects

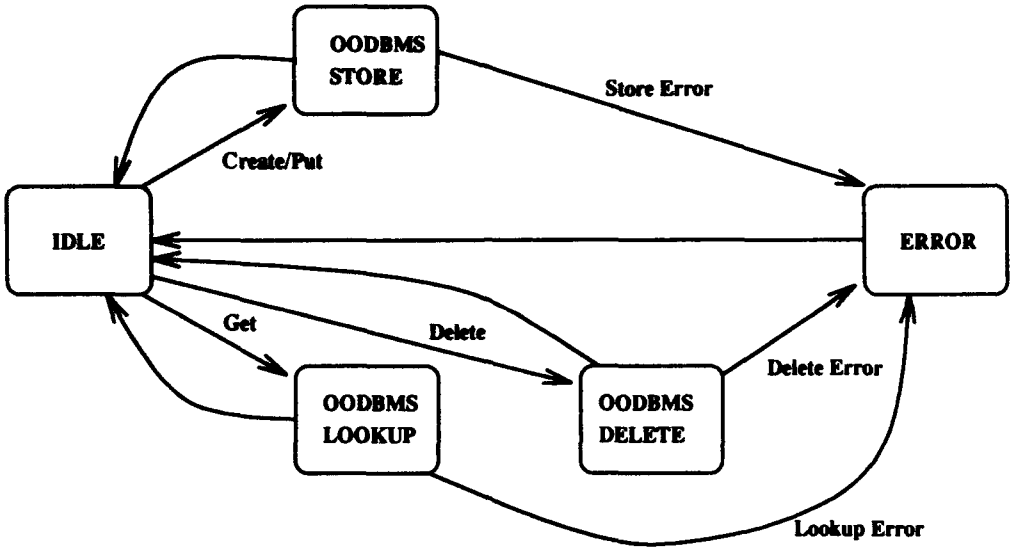

Figure 12. State Diagram for Persistent Objects

Persistent objects offer the application complete control over when transaction processing takes place. Thus, when issuing orders to form aircraft packages, we can lock each airbase as aircraft are allocated from them until the entire package is formed and committed. This prevents a user on another workstation from allocating the same aircraft again.

The major disadvantage of this approach is the impact on existing Saber code. Each object class in the simulation would have to be rewritten to activate persistence. Since one goal of this thesis was to make minimum impact, this option was not entirely desirable.

3.4.2.3 Flat-File Simulation. The option having the least impact on the application is to just replace the read and write methods of each object with implementations that access the database. Data is still read into and written from in-memory data structures. In essence this makes the OODBMS look like the set of flat files to Saber.

The major drawback to this approach is that the good features of an OODBMS are never effectively used. This results in large amounts of data being loaded into memory, even though only a small fraction of it will actually get accessed. The hexagon ground grid fits precisely into this category, since in a 100 by 100 hex map with 200 units, if each unit moves 1-3 hexes (75km), only 600 hexes out of 10,000 would be accessed. Air packages

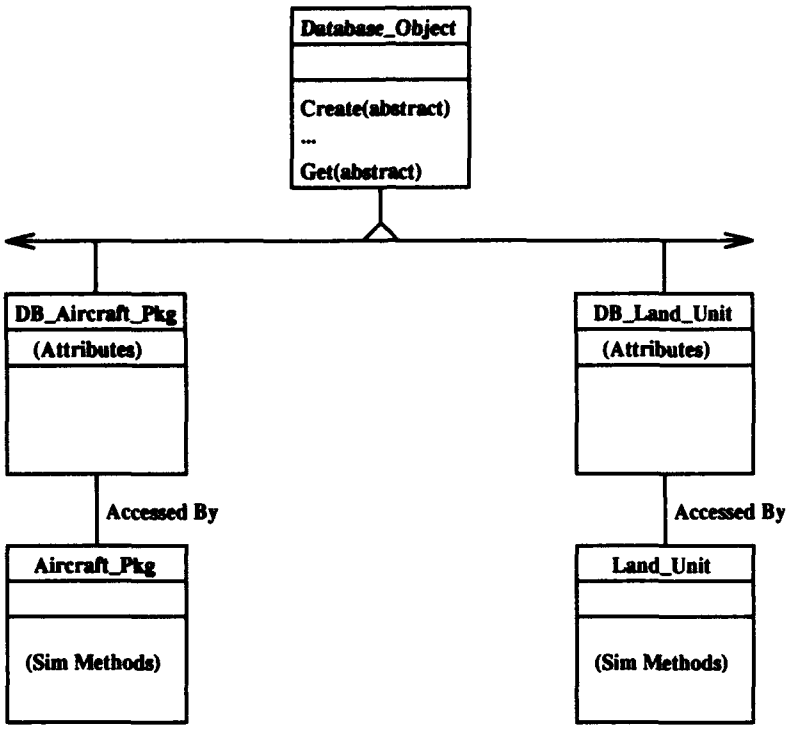

Figure 13. Object Model for Flat-File Simulation

might increase this number, but not by much. Thus, 80 to 90 percent of the map need never be loaded.

One advantage over flat files, however, comes in terms of speed. The OODBMS can perform a linear traversal of a data structure so that loads can be performed in $O(n)$ time. For small scenarios the flat files will actually perform faster than the OODBMS. For medium to large scenarios, however, the OODBMS will be faster. The next chapter discusses the differences in load times between a 10 by 10 hexagon map and a 100 by 100 hexagon map for flat files and the OODBMS interface.

The design of the replacement I/O routines is relatively straightforward. First, we create storage object classes for each of the object classes in Saber. These storage classes contain the stored attributes of an object and the methods to move them in and out of the database. They are derived from the database object class. Figure 13 shows the object model of this method.

Next, we alter the existing application object's class diagram to show the relationship they have with the database objects. The dynamic nature of each application object is altered to show that loading and saving is now performed by interfaces to the database objects. The dynamic behavior of the database objects in this method are identical to that shown previously with persistent objects. The difference is that the application object controlling a database object invokes all database-altering methods.

3.4.3 Design of Saber Application Classes. In order to connect the user interface to the simulation engine, they had to agree on how data structures should be accessed in the OODBMS. It wouldn't be possible to have the simulation engine using a persistent array for the hexagon map and have the user interface use flat file simulation—the database objects would have different attributes. To create the most straight-forward implementation, and reduce the amount of impact on the application code, flat file simulation is used for most of the Saber database access routines.

As discussed previously, using flat file simulation requires that a storage object class be created for each application object class that must be saved. Ada provided a natural way to formulate this part of the design via the package specification. A package for each storage class was created using Ada syntax. This provided a convenient way to specify the functional interface without dedicating it to a specific implementation (beyond the fact it would be written in Ada).

During implementation, it was only necessary to create the body for each specification. This method of transition from design to implementation was quite successful as the only change to the specifications occurred when it was decided to change the implementation to a generic package. This required the addition of an instantiation clause into the specification.

3.5 Summary

In this chapter we discussed how the database interface requirements were analyzed, how the interface was designed, and how this design was utilized for Saber. The interface requirements were determined using OMT. These were generalized so that the interface

would work not only with Saber, but with a variety of simulation domain applications. The interface was designed to support persistent data types, persistent objects, and flat-file simulation. Since flat-file simulation has minimal code impact on Saber, it was chosen as the best design approach given the time constraints of the research. A more robust approach would have been to make the design changes to Saber needed to implement the simulation using persistent objects.

IV. Database Implementation

4.1 Overview

This chapter discusses how we implemented the design in Chapter III. It shows how the SAIC OODBMS is organized, how the abstract database interface was created using Ada83, and how the OODBMS was tied to the interface. We show code demonstrating how persistent structures are developed using the interface and the specific changes required to Saber application code.

4.2 SAIC Object-Oriented Database

The SAIC OODBMS was described in Chapter II. This section discusses some of the implementation-specific features of the OODBMS. The source code for the SAIC OODBMS is divided into two main areas: packages which comprise the server, and interface packages which both the application and the server use.

4.2.1 Server. The server is a separate executable that fields all database requests from clients. This task is done in the following manner (see Figure 14):

1. Client Request. The server, after initialization, enters a wait state until a client submits a request. This is done by executing routines in a Unix System V binding package that control a system semaphore. When the semaphore is set, the server is notified and looks for a request packet.
2. Client Communications. All communication between the server and client is through shared memory segments. Thus, when a client request is indicated, the server looks in its dedicated segment to get the segment id of the client. It then allocates that memory and is able to work in it. The drawback to using this approach is that clients cannot execute on remote nodes and must log into the server's host.
3. Method Resolution. The client request will typically contain a method to be executed by an object. The request is routed to an appropriate object class resolver by a method resolution package—basically a large case statement. Once in an object

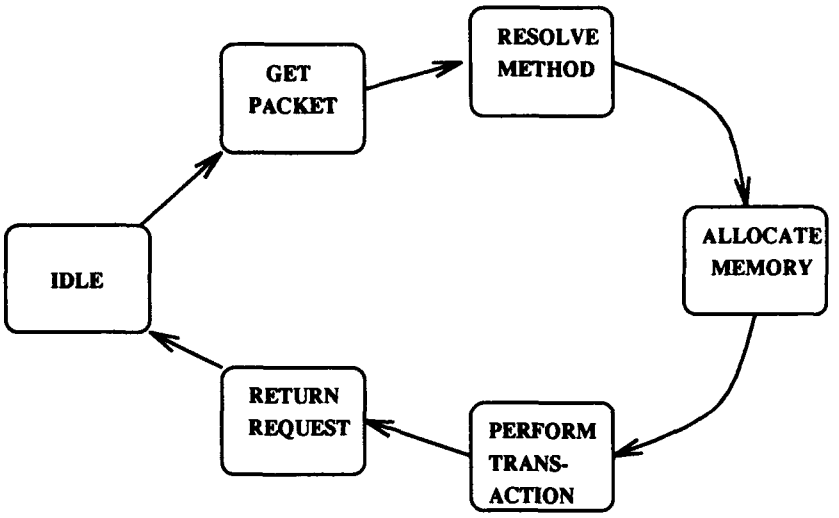

Figure 14. SAIC OODBMS Client Request Handling

class' resolver package, the method will be executed by routing it to an appropriate routine via another case statement. If the method is not defined for this object, it is routed to the parent of this object class instead.

4. Virtual Memory. As the method executes, it will need to act upon the database's stored objects. The database is basically treated as virtual memory, dividing a single storage file into pages. Each of these pages is treated like a memory heap. When allocations are requested, the system searches for a free page in memory with enough space for the allocation. Failing this, it looks for a page on disk with enough space. If this fails, an additional page of memory is generated and the virtual memory file (the database storage file) is lengthened.

5. Transaction Handling. Some client requests will signal the start or end of a transaction. When the database starts a transaction, new virtual memory is allocated to work in. If the transaction commits, the allocated pages are written to the actual database's virtual memory. If the transaction aborts, the pages are discarded.

6. Request Results. As the request is processed, the results are written into the client's shared memory segment. If a transaction returns large amounts of data, there is a good possibility of overflowing the segment. To prevent this, either the communications segment must be enlarged or the client must break up requests into smaller

pieces. When the request is complete, the Server resets the system semaphore and waits for the next request.

4.2.2 Interface. The interface to the SAIC OODBMS is not trivial. There are several packages that must be used in order to get the communications to work, to get the requests to transfer, and to get the results into a recognizable form.

4.2.2.1 Communications Package. The Communications_Buffer package delivers the utilities needed to establish an interface with the database server. This is accomplished by allocating a shared memory segment and making it accessible by the server. This package also contains several needed types and conversions for extracting data from the allocated memory buffer.

4.2.2.2 Interface Package. The Database_Interface package contains procedures and functions need to create, open, and close the database. It also contains routines for initiating the start of a transaction and committing or aborting it.

4.2.2.3 Predefined Classes Package. Each object class in the SAIC database has a declaration in the Predefined_Classes package. This allows an object of that class to be created—the system uses the identifiers as a way to send messages to a class (like the "new" message). Additionally, if an object is sent a "class" message, it will respond with an identifier from this package.

4.2.2.4 Database Types Package. The Database_Types package is significant in that it contains the declaration for an Object_Id, a unique 32-bit identifier assigned to each database object.

4.2.3 Corrections to OODBMS Code. As provided by SAIC, the OODBMS did not run properly. The server would abort execution due to a SAIC-defined "UNABLE TO ALLOCATE HEAP" exception.

Tracing this problem using the SPARCworks debugger, it was isolated to the point at which the system would attempt to allocate a virtual memory heap. This heap was

mapped into real memory using an overlay technique. Heap status data was then copied into this area. When the system would verify the heap (a rather odd thing to do in a supposedly working program), the status data would be gone.

A common complaint about Ada is its unfriendly handling of overlays, and this turned out to be the cause of the problem. SunAda 1.0, the version used by SAIC, did not enforce the Ada rule that pointers always be initialized to the value NULL. SunAda 1.1 did begin enforcement of this rule. Thus, when the verify operation overlayed a record template onto the memory area, Ada initialized all the pointers in the record to NULL, wiping out access to necessary heap data.

According to Appendix F of the SunAda Programmer's manual (23), the solution is to use a 32-bit integer in place of the pointer. All references to the pointer must then use Unchecked_Conversion to force the value to a pointer type. Executing this change caused the system to work correctly. The offending code and its change are shown in Figure 15.

4.3 Database Interface

The implementation of the abstract interface in Ada occurred in an evolutionary manner. Several implementations of the design were attempted and rejected. This section describes the version completed at the time of this writing. Alternatives and suggestions for improved versions are discussed here and in Chapter VII. That several versions could be tried and improved upon without impact to the application object classes indicates the correct level of abstraction has been obtained.

4.3.1 Organization. The interface is organized into three main packages: IM_Interface, IM_Internals, and IM_Generic. Figure 16, a dependency graph, shows how the packages fit into an application.

4.3.1.1 IM_Interface. The purpose of this package is to provide general use functions that are not specific to any object class. Examples: Open_Database, Start_Transaction, Rollback_Transaction, etc. Additionally, this package supplies a declaration IM_Interface.ObjID which refers to the unique ID attached to each database object.

```
-- Old Version
   type HEAP_MANAGEMENT_RECORD is
      record
         Check        : INTEGERS;
         Length       : OFFSETS;
         The_Integers : INTEGER_ARRAY;
         Last_Hole    : NATURAL;
         The_Holes    : HOLE_ARRAY;
         Double_Check : INTEGERS;
         ChangePtr    : CHANGE_PTR;     -- This will be initialized
                                        -- to zero each time an
                                        -- overlay occurs.
         Tracking     : BOOLEAN;
      end record;

-- Corrected Version
   type HEAP_MANAGEMENT_RECORD is
      record
         Check        : INTEGERS;
         Length       : OFFSETS;
         The_Integers : INTEGER_ARRAY;
         Last_Hole    : NATURAL;
         The_Holes    : HOLE_ARRAY;
         Double_Check : INTEGERS;
         ChangePtr    : INTEGER;        -- An INTEGER occupies the same
                                        -- space as a pointer.  It will
                                        -- not be initialized.
         Tracking     : BOOLEAN;
      end record;
```

Figure 15. SAIC Heap Overlay Record: Old and New Versions

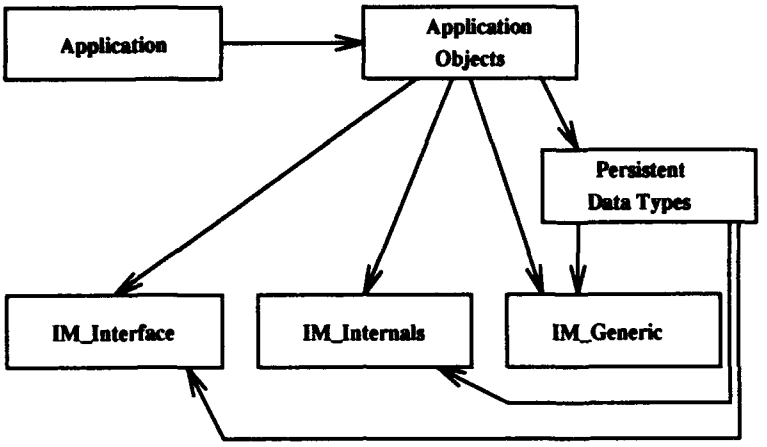

Figure 16. Interface Packages Dependency Graph

4.3.1.2 IM_Internals. Originally intended to be hidden from the application, this class would have been better named "IM_Objects." It contains the enumerated type that identifies all the possible application object classes. Applications must register their classes by adding new values to the enumeration.

4.3.1.3 IM_Generic. This package is the database object class for the interface. It contains generic functions and procedures that allow objects in the database to be manipulated. By instantiating this class and the generic functions within it, application object classes are formed as children of this abstract class.

4.3.2 Interface Package. The interface package implements the following subprograms and declarations:

- Create_Database. This procedure creates the database named in the string Database_Name. This is implemented in the SAIC database by passing the string to the Database_Interface.Create_Database procedure. This procedure sends a request to the server to make a copy of the file oo_dbms.base, the template file that the database initially builds. The new database is then opened for use.
- Open_Database. This procedure opens the database named in the string Database_Name. This is implemented by passing the string to the SAIC Database_Interface.Open_Database procedure.

- Copy_Database. This procedure allows the database to be copied for archival or versioning purposes. It is implemented by passing two strings into the Database_Interface.Create_Database function. Instead of copying from oo_dbms.base it copies from the database named in the string Old_Database.
- Close_Database. This procedure closes the currently open database.
- Start_Transaction, Commit_Transaction, Rollback_Transaction. These procedures map directly into the procedures by the same name in Database_Interface. When Start_Transaction is invoked, all other transactions on the database are blocked—a limitation of the SAIC OODBMS. Only a Commit_Transaction or Rollback_Transaction will release them.
- Package Wrapper, Wrapper_Types. These packages are a concession to the use of the SAIC database. The application instantiates the wrapper package by naming its main procedure as the application routine. The wrapper then allocates the communications buffers and initiates communications with the server. This code was written by SAIC and has been moved into the IM_Interface package to eliminate the need for applications to see the database.

  In another database, the wrapper might not be necessary. In that case its body would be empty other than a call to the application's main procedure. A better implementation might be to create new procedures in IM_Interface that handle client initialization and shutdown.

4.3.3 Generic Specification. The basic idea behind the package IM_Generic is that application objects will inherit behavior by instantiating this package. Basic functionality for creating, storing, retrieving, and deleting objects then become available. Further functionality is obtained by instantiating the generic functions in the package to allow creation and manipulation of object relationships. The following functions and procedures are available in the package:

- Create. This function implements the Create method defined in the database object. It takes a value of the user-defined type and embeds it in a database node structure.

This structure is then stored in the database as a binary large object (BLOB) and the object identifier assigned to it is returned to the user. Create is overloaded to allow an integer or string key to be associated with the value. The keys are stored in a dictionary (a set of key-object_id pairs) for fast retrieval.

- Get_ObjID. Given a key value, this function will search the class' dictionary for a matching key. If one is found, the id of the object containing the associated value is returned. This object id may then be used in the Get routine to obtain a value. If no matching key is found, a null object id (as defined in IM_Interface) is returned.
- Get. This procedure takes an object id and returns the value of the object it references. An invalid object id will generate a constraint error exception.
- First. This class method returns the first object in the class. This is done by sending a "first" message to the interface-defined set in which this class of objects is stored. If the set is empty, a null object id is returned and the Status parameter is set to False. Otherwise, the id of the first object in the set is returned and the Status parameter is set to True.
- Next. Complementary to the First procedure, this procedure finds the next object in a class. This is accomplished by sending a "succ" (successor) message to the object referenced by the object id in the Cur_Object parameter. If a successor exists, the object id is returned in New_Object and Status is set to True. If a successor does not exist, Status is set to False.
- Delete. Removes an object and all its associated sets from the database. If an invalid object id is passed a constraint error exception will be raised.
- Add_To_Set. This generic procedure requires that it be instantiated with the name of the relationship that the set represents. This name is taken from an enumerated range that is passed when the generic package is instantiated. When called, the procedure adds the object id in Tgt_Object to the set.
- Remove_From_Set. The opposite of Add_To_Set, this procedure removes the object id in Tgt_Object from the set. If the value does not exist, a constraint error exception is raised.

- First_In_Set. This procedure returns the first object id contained in a relationship set. The value is returned in Tgt_Object. If the set is empty, Status is set to False, otherwise it is set to True.
- Next_In_Set. This procedure finds the object id contained in Cur_Object and returns the object id that follows it in Tgt_Object. Using this procedure with First_In_Set allows an application to iterate through the relationship set.
- Add_Object. Like the set operations, this generic procedure also requires that it be instantiated with the name of the relationship that it represents. Unlike a set, which implements a one-to-many relationship, this object pointer implements a one-to-one relationship and will only contain one value at any time. This procedure stores the object id contained in Tgt_Object into the object pointer.
- Get_Object. This procedure returns the value of an object pointer in an object.
- Extract_Set. This function allows a simple query to be performed on a class of objects. It is instantiated with two functions, one to extract a field from a database object's attributes, the other to compare it with the value passed in The_Target. The function iterates through the set looking for matches. The object id of matching objects are stored in a non-persistent set and the object id of this set is returned.
- First_In_Extract_Set. Analogous to the First_In_Set operation, this procedure works on a set extracted with Extract_Set.
- Next_In_Extract_Set. Like Next_In_Set, this procedure works on an extracted set.
- Delete_Extract_Set. This operation frees the non-persistent memory associated with an extracted set.

4.4 Persistent Structures

The next level of abstraction up from the interface is the persistent data type. The implementation of two persistent data types, linked-lists and arrays, are discussed in this section.

4.4.1 Linked-Lists. Persistent linked-lists are easily developed using the interface. For Saber, the doubly-linked-list developed by Major Eric Christensen (based upon the Booch components (5)) was converted. The object was to keep the system interface, i.e., the package specification, the same regardless of whether or not the list implementation is persistent. A comparison of an example procedure body before (Figure 17) and after (Figure 18) conversion is instructive.

First, a new object class is created from the base database object class. It operates on a set that contains object ids for all of the persistent data types that will be used in the system. In essence, it is a container class for heterogeneous objects. By instantiating a version of it with a linked-list node structure for a data type, we can store linked list elements in it. These elements will coexist with other elements inserted by different instantiations of this package. In Figure 17, List_Manager is an instantiation of a memory management package. In Figure 18, List_Manager is an instantiation of the container class for linked-list nodes.

The two procedures have very similar code. The main exceptions occur when a database item must be read or written. Memory items do not have this requirement. For example, in the procedure Construct_Front the old head of the list DB_Node has its Previous attribute set to the new head of the list, Temporary_Node. A List_Manager.Put operation is then required to make the change persistent.

4.4.2 Arrays. Arrays are a much more difficult matter to implement in Ada83. Ada cannot overload the assignment operator (":="). This precludes writing a method by which a procedure can be invoked to modify a persistent object when an assignment is made. Similarly, it is not possible to overload the parenthesis operators ("(<*array expression*>)") to have them return a persistent value when a data type is used in an expression.

The only alternative is to use syntax similar to that for linked-lists. As discussed in the previous chapter, the operations required would be much simpler: Get and Put. Due to the impact of the change, however, it was decided that persistent array usage in Saber would be impractical given the time constraints of this work.

```
procedure Construct_Front(The_Item     : in Item;
                          And_The_List : in out List) is
  Temporary_Node : List;
begin
  if And_The_List = null then
    Temporary_Node := List_Manager.New_Item;
    Temporary_Node.Previous := null;
    Temporary_Node.The_Item := The_Item;
    Temporary_Node.Next := null;
    And_The_List := Temporary_Node;
    List_Manager.Free(Temporary_Node);
  elsif And_The_List.Previous = null then
    Temporary_Node := List_Manager.New_Item;
    Temporary_Node.Previous := null;
    Temporary_Node.The_Item := The_Item;
    Temporary_Node.Next := And_The_List;
    And_The_List := Temporary_Node;
    And_The_List.Next.Previous := And_The_List;
    List_Manager.Free(Temporary_Node);
  else
    raise Not_At_Head;
  end if;
exception
  when Storage_Error =>
    raise Overflow;
end Construct_Front;
```

Figure 17. Original Doubly-Linked-List Procedure

```
procedure Construct_Front(The_Item     : in Item;
                          And_The_List : in out List) is
  Temporary_Node : Node;

begin
  if Check_Init(And_The_List) then
    if And_The_List = IM_Interface.Null_ObjID then
      Temporary_Node.Previous := IM_Interface.Null_ObjID;
      Temporary_Node.The_Item := The_Item;
      Temporary_Node.Next := IM_Interface.Null_ObjID;
      List_Manager.Put(And_The_List, Temporary_Node);
    else
      List_Manager.Get(And_The_List, DB_Node);
      if DB_Node.Previous = IM_Interface.Null_ObjID then
        Temporary_Node.Previous := IM_Interface.Null_ObjID;
        Temporary_Node.The_Item := The_Item;
        Temporary_Node.Next := And_The_List;
        New_Id := List_Manager.Create(Temporary_Node);
        DB_Node.Previous := New_Id;
        List_Manager.Put(And_The_List, DB_Node);
        And_The_List := New_Id;
      else
        raise Not_At_Head;
      end if;
    end if;
  end if;
exception
  when Storage_Error =>
    raise Overflow;
end Construct_Front;
```

Figure 18. Converted Doubly-Linked-List Procedure

4.5 Changes to Existing Saber Code

The flat-file simulation method of implementing the database was the most attractive approach for integrating the system. All that was required was to add new code for reading and writing the database objects. This held an advantage over persistent data types and persistent objects where major code impact would occur (see note previously about persistent arrays). This would prove to be a costly performance mistake in some areas, and a boon in others. This section describes how that change was made and how the others could have been done selectively to provide dramatic performance boosts.

4.5.1 Flat-File Input/Output Changes. Each object class in the simulation engine contained code for reading its object data from a flat file. When the simulation started, a main controller would invoke the read method on the object class. Upon completion, a similar write method was invoked. An example of this type of code is shown in Figure 19. This piece of code reads in the trafficability of a hexside for a given hex. Note the code is an $O(n^2)$ algorithm since the trafficability file must be traversed each time a hex is read.

This code was replaced as described in the design section—database object classes were created and code was written to replace the flat-file routines as follows:

1. *Database Class Creation.* Ada package specifications had been developed during the final design phase of the Saber database interface. Each package instantiated the database interface generic package with the methods needed to manipulate the object. Since all the attributes of an object are encapsulated within a package, this forces the simulation and user interface to adopt the same data types when extracting information.
2. *Code Replacement.* Since it was convenient to be able to read flat files to initialize the object database, the original input routines were simply renamed to indicate they read flat files. The actual database input/output routines were then inserted. An additional function was added to each class that caused it to read from a flat file and immediately write back to the database. This allowed the construction of a simple database initialization program.

```
procedure Get_Hex_Pie_Pieces (The_Pie_Pieces : in out Pie_Array_Type;
                              The_Sides : in out Sides_Array_Type;
                              The_Lon : Hex.Lon; The_Lat : Hex.Lat) is

Travel_Id : Integer;
The_Input : Text_Io.File_Type;
Pie_Piece : Hex.Direction_Type;
Travel_Lat : Hex.Lat := 1;
Travel_Lon : Hex.Lon := 1;
Travel_Level : Hex.Level := 1;

begin
  Text_Io.Open (The_Input, Text_Io.In_File, Input_File_Hex_Pie_Pieces);
  -- skips 2 comment lines at the beginning of the file
  Text_Io.Skip_Line (The_Input,2);
  while not Text_Io.End_Of_File (The_Input) loop
    Text_Io.Set_Col (The_Input, 3);
    Int_Io.Get (The_Input, Travel_Id);
    Hex.Convert_No (Travel_Id, Travel_Lon, Travel_Lat, Travel_Level);
    if (The_Lat = Travel_Lat) and then (The_Lon = Travel_Lon) then
      Text_Io.Set_Col (The_Input, 10);
      Hex.Dir_Io.Get (The_Input, Pie_Piece);
      Text_Io.Set_Col (The_Input, 15);
      Int_Io.Get (The_Input, The_Sides(Pie_Piece).Hexside_No);
      Text_Io.Set_Col (The_Input, 22);
      Hex.Traffic_Io.Get (The_Input, The_Pie_Pieces(Pie_Piece));
      Get_Feba(The_Sides(Pie_Piece));
      Get_River(The_Sides(Pie_Piece));
    end if;
    Text_Io.Skip_Line (The_Input);
   end loop;
   Text_Io.Close (The_Input);
end Get_Hex_Pie_Pieces;
```

Figure 19. Flat-File Input Routine for Hex Side Trafficability

```
procedure Get_Hex_Pie_Pieces (The_Pie_Pieces : in out IM_Pie_Piece.Pie_Array_Type;
                              The_Sides : in out IM_Hex_Side.Sides_Array_Type;
                              Hex_Id    : in IM_Interface.ObjID) is

  Temp_Id : IM_Interface.ObjID;

begin
  IM_Ground_Hex.Get_N_Side(Hex_Id, Temp_Id);
  IM_Hex_Side.Get(Temp_Id, The_Sides(IMHT.N));
  IM_Ground_Hex.Get_NE_Side(Hex_Id, Temp_Id);
  IM_Hex_Side.Get(Temp_Id, The_Sides(IMHT.NE));
  IM_Ground_Hex.Get_SE_Side(Hex_Id, Temp_Id);
  IM_Hex_Side.Get(Temp_Id, The_Sides(IMHT.SE));
  IM_Ground_Hex.Get_S_Side(Hex_Id, Temp_Id);
  IM_Hex_Side.Get(Temp_Id, The_Sides(IMHT.S));
  IM_Ground_Hex.Get_SW_Side(Hex_Id, Temp_Id);
  IM_Hex_Side.Get(Temp_Id, The_Sides(IMHT.SW));
  IM_Ground_Hex.Get_NW_Side(Hex_Id, Temp_Id);
  IM_Hex_Side.Get(Temp_Id, The_Sides(IMHT.NW));

  IM_Ground_Hex.Get_N_Pie_Piece(Hex_Id, Temp_Id);
  IM_Pie_Piece.Get(Temp_Id, The_Pie_Pieces(IMHT.N));
  IM_Ground_Hex.Get_NE_Pie_Piece(Hex_Id, Temp_Id);
  IM_Pie_Piece.Get(Temp_Id, The_Pie_Pieces(IMHT.NE));
  IM_Ground_Hex.Get_SE_Pie_Piece(Hex_Id, Temp_Id);
  IM_Pie_Piece.Get(Temp_Id, The_Pie_Pieces(IMHT.SE));
  IM_Ground_Hex.Get_S_Pie_Piece(Hex_Id, Temp_Id);
  IM_Pie_Piece.Get(Temp_Id, The_Pie_Pieces(IMHT.S));
  IM_Ground_Hex.Get_SW_Pie_Piece(Hex_Id, Temp_Id);
  IM_Pie_Piece.Get(Temp_Id, The_Pie_Pieces(IMHT.SW));
  IM_Ground_Hex.Get_NW_Pie_Piece(Hex_Id, Temp_Id);
  IM_Pie_Piece.Get(Temp_Id, The_Pie_Pieces(IMHT.NW));

end Get_Hex_Pie_Pieces;
```

Figure 20. OODBMS Input Routine for Hex Side Trafficability

Figure 20 is the code which replaced Figure 19. Note that the algorithm is $O(n)$. It simply reads a hex, then calls this routine to look up the trafficability of the hexside.

4.5.2 Problems. Two problem areas were encountered while implementing the database interface and integrating it with Saber. These areas, performance and consistency, are discussed in the following sections.

4.5.2.1 Performance. "Bolting-on" an OODBMS worked well for the simulation engine. Testing against the full scenario, the flat file reads had taken over 6 hours to read only 800 hexagons on a 10000 hex grid. The test was stopped since it appeared that the algorithm would require several days to read the entire scenario (the $O(n^2)$ performance factor). Using the OODBMS, these same 800 hexagons were read in 5 minutes.

Unfortunately, the user interface, not requiring all of the data needed by the simulation engine, could read all 10000 hexes from the flat files in under a minute. The same operation for the database routines required nearly 60 minutes. This was unacceptable performance since the user interface should be available on demand to users.

The temporary solution to this problem was to create a "compatibility" class in the database interface packages. Upon completion of a run, the simulation invokes a method in this class which writes a file from which the user interface can directly read hexagon data. This works since the user interface never alters the hex attributes. All other data required by the user interface is still read from the database.

A permanent solution to this problem would be a re-design of the user interface to utilize persistent objects or persistent data types. This would eliminate the need for the up-front loading. A possible problem with this solution is the performance of the system as the user scrolls around the map. It might be necessary to lock hexagons read from disk into memory and then perform look-ahead reads so that hexes are available as scrolling occurs. Eventually, this would end up loading the entire map.

4.5.2.2 Consistency. One of the immediate problems encountered in integrating the user interface and simulation was that of data type consistency. As discovered during the design and analysis phases, the attributes of each object class matched for both

subsystems. During implementation, however, it was discovered that their physical representation in memory did not. For example, the simulation stores hex numbers as integers. The user interface stores them as characters.

This problem was overcome by making an ad-hoc rule that "the simulation format is correct." This assumption was made since the simulation had been reconstructed and had a better object-oriented structure than the user interface. As will be discussed in the next chapters, it is clear that the user interface will require serious design changes. For now, the format is simply converted to what the user interface expects after it is extracted from the database.

4.6 Summary

This chapter examined how the database interface design from Chapter III was implemented in Ada. Three basic interface packages were constructed: IM_Interface, IM_Internals, and IM_Generic. IM_Interface provides general utilities needed to operate the database. IM_Internals is used to enumerate the object classes that will be stored in the database. The IM_Generic package is instantiated to create a database object class. Persistent data types, persistent objects, and flat-file simulation were discussed. Flat-file simulation was chosen due to minimal code impact and the time constraints of the project.

V. User Interface Analysis and Design

5.1 Overview

The Saber user interface displays a map, overlayed with hexagons, on the screen and allows the player to manipulate the map, units displayed on it, and their orders. It is designed to be friendly and intuitive. A sample of the Korean scenario is shown in Figure 21.

Basic functionality for building user input forms was missing, however. In order to complete Saber, it was necessary to develop these low-level building blocks and complete a sample mission input dialog to prove they worked as expected. Since air combat was not complete in the simulation, we decided to implement the land mission input dialog.

In this chapter we describe how the requirements for the interface were analyzed and how a designed was created to satisfy them.

5.2 Problem Analysis

There were several obstacles that we needed to be solve before a working user interface could be developed. First, the original interface had been developed using the STARS bindings—outdated and unsupportable (14), they had to be converted to something better supported. Second, the mission input forms had to be developed so that the user could issue orders to the units. Last, the operation of the user interface had to be integrated such that it appeared seamless with the simulation engine.

5.2.1 Conversion to New Bindings. The original STARS bindings were developed under Motif v1.0. Since that time, Motif has progressed to version 1.2, and the underlying X libraries have undergone three major releases. The result of this is that Saber had become extremely outdated and nearly unsupportable on machines with newer X libraries.

The remedy for this situation was to replace the STARS bindings with a newer, commercially supported set. In this case, that set was the SERC bindings described in Chapter II.

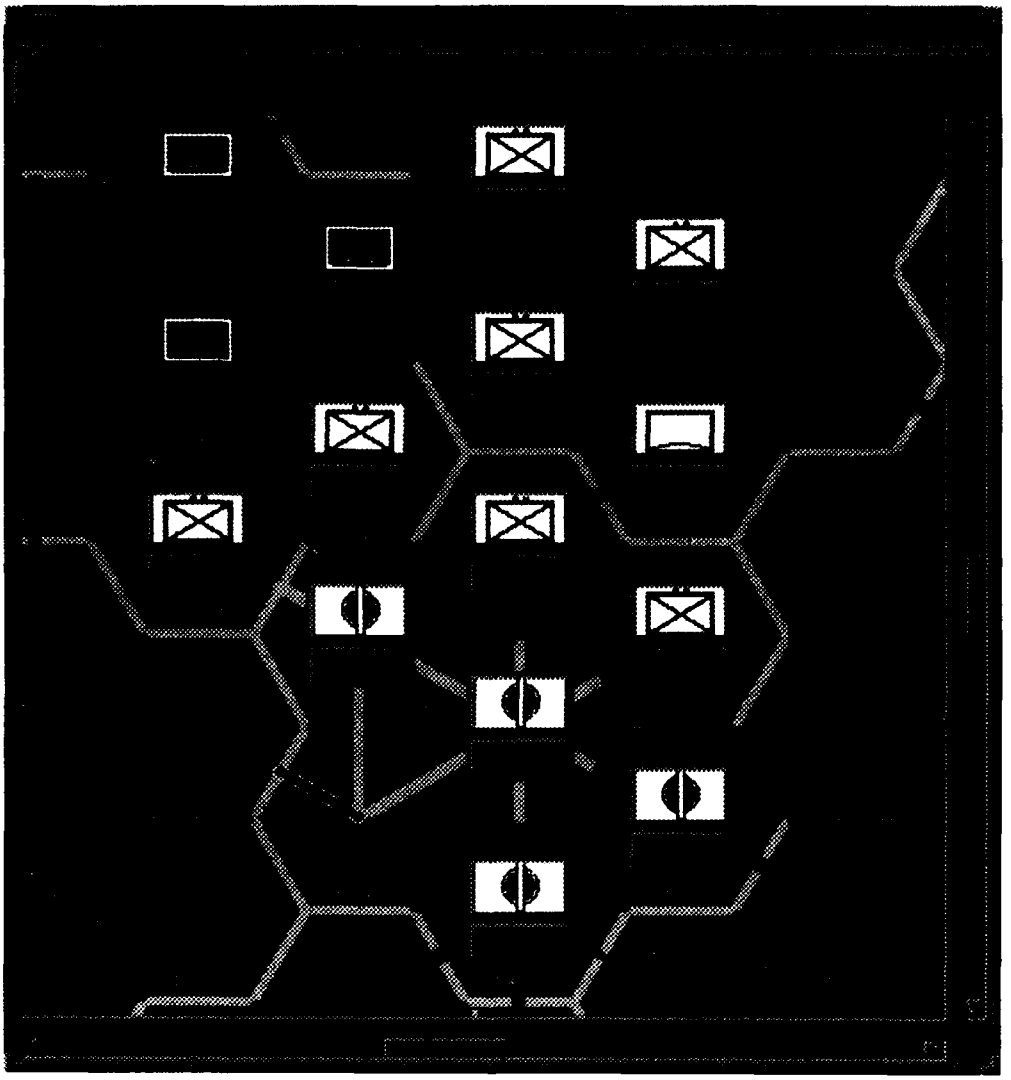

Figure 21. Saber Game Board Map

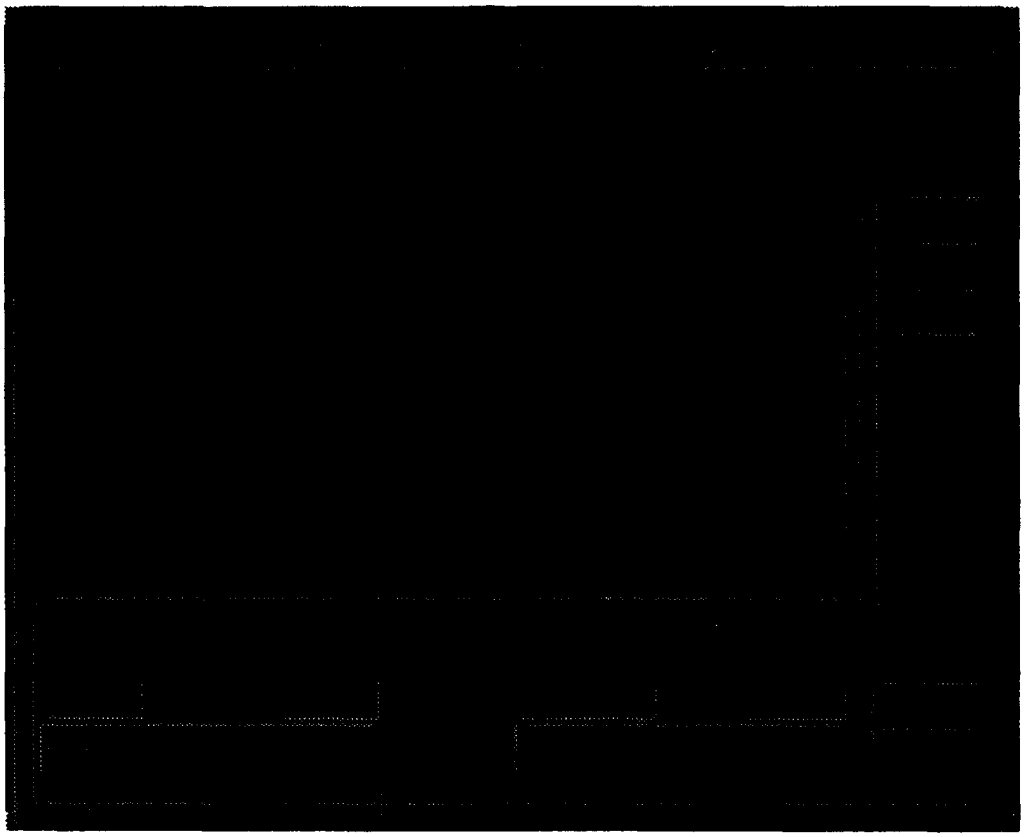

Figure 22. Saber Beddown Mission Entry Form

5.2.2 Analysis of Mission Entry Forms Requirements.

5.2.2.1 Previous Work. The mission entry forms required a high degree of consistency to remain user friendly. Earlier work by Horton (9:134-136) and Moore (14) attempted to address this requirement by developing some preliminary versions of the air bed-down missions form. Unfortunately, the developed form cannot be used since it does not comply with the Motif style guide, and therefore is not consistent with the other dialogs in the system. A simple rearrangement of the form is very difficult since the Ada code is not properly formatted and the layout is hard-coded into the application. The form is shown in Figure 22.

5.2.2.2 Identification of Objects. Figure 23 is a representation of how the land unit movement input form should be built to comply with the Motif style guide

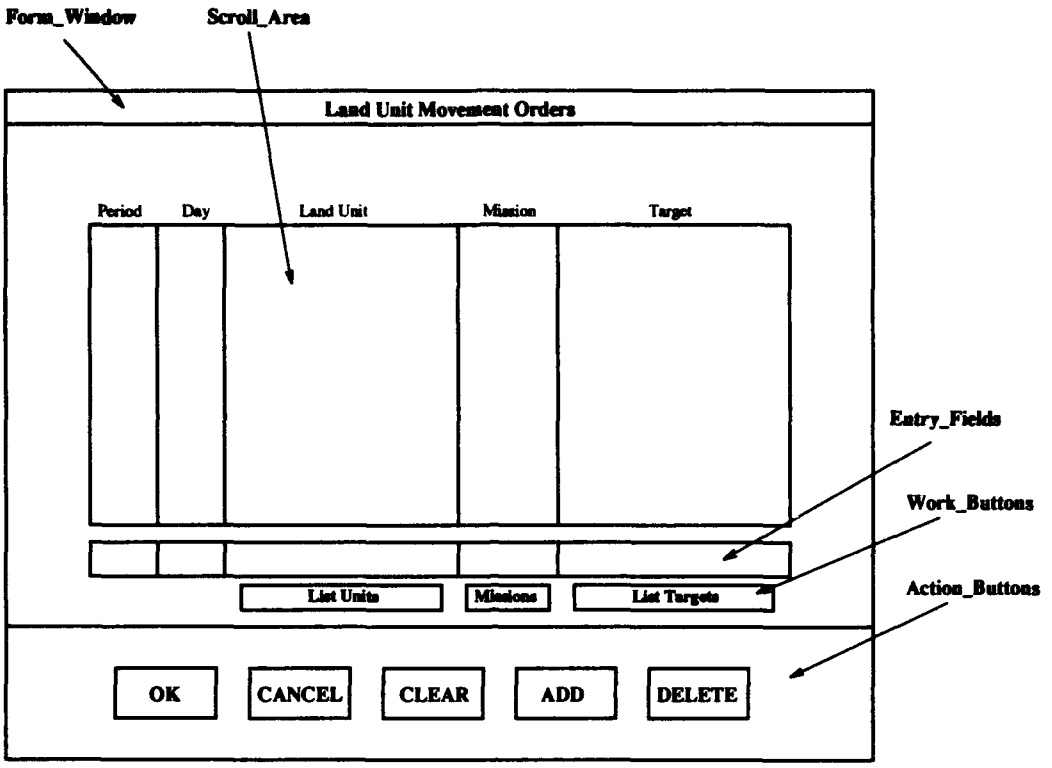

Figure 23. Land Unit Movement Order Entry Form

recommendations. The form is divided into two areas—the work area and the action area. The work area consists of the scrolled list of orders, the input fields, and all supporting help buttons and labels. The action area is always at the bottom of the form and consists of the Save, Cancel, and Clear buttons.

Figure 24 represents how the aircraft movement (bed-down missions) form would be layed out. Note the similarity of both forms not only in layout, but in the components used. This illustrates the need for development of a set of low-level objects for Motif that Ada can use to build input forms. Figure 25 shows the "generic" components of a typical input form. Components layed out in this manner will conform to the Motif style guide's (18:7-16,7-17) recommendations.

By examining the various forms that are required for Saber (see Horton (9)), the various components needed can be identified:

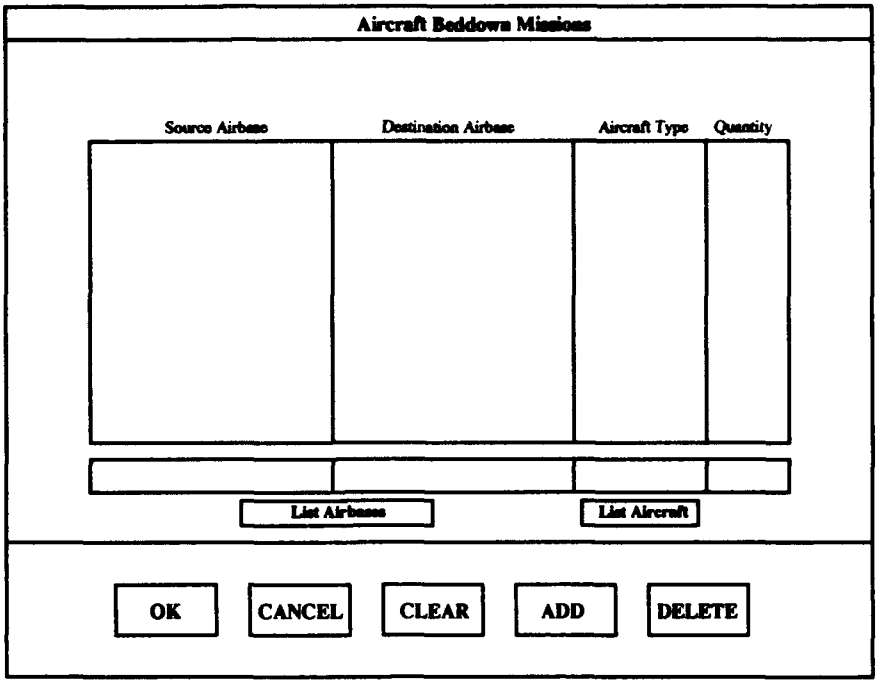

Figure 24. Aircraft Bed-down Mission Order Entry Form

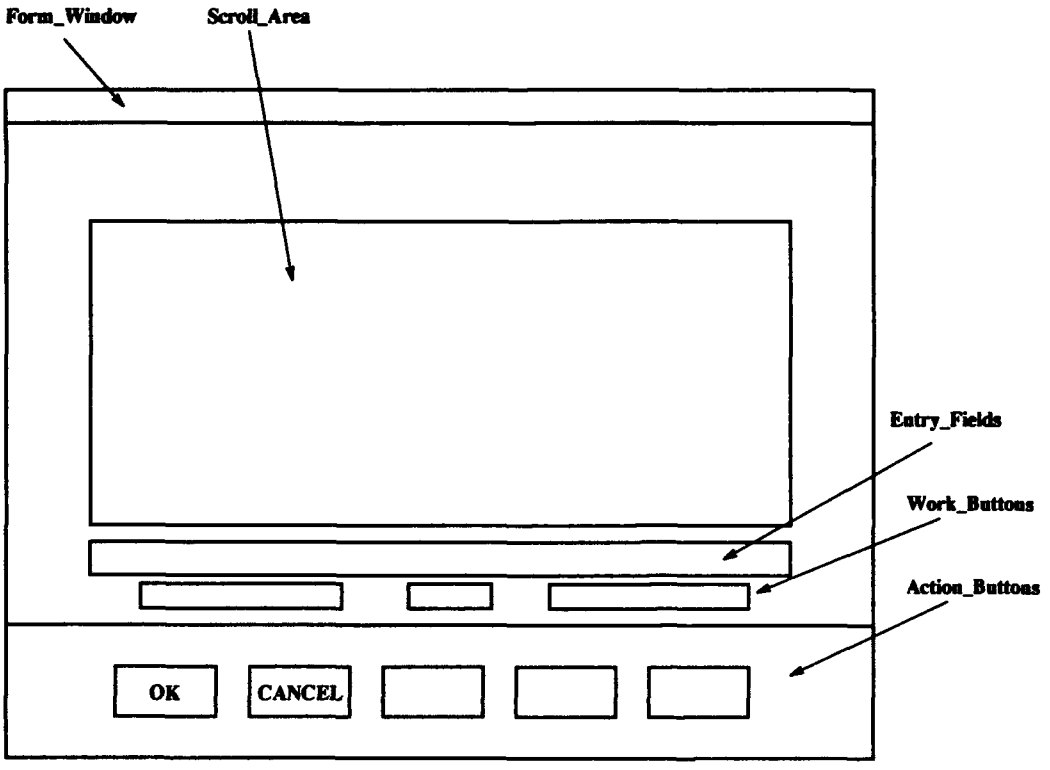

Figure 25. Generalized Mission Entry Form

- Form_Window. This is the background window (also known as a dialog (8)). All other components are placed on this window to create the input form.
- Action_Buttons. This component consists of several buttons that exercise control over the entire form. Usually the Save and Cancel buttons are mandatory. This component will automatically be located at the bottom of the form and will place a separator line immediately above itself.
- Scroll_Area. Each input form displays information already input in a scrolled list. This component would automate the construction of that list. It would be placed at a location specified by the developer.
- Entry_Field. This component would allow the entry of a piece of information. Each field would have an optional validation routine associated with it, as well as optional help and list-of-values buttons.
- Work_Button. This button would cause a specified routine to activate when pressed. Its purpose is to allow for features such as delete an entry, add and entry, etc.
- Label. This is simply a piece of text that may be placed anywhere on the Form_Window.
- Values_List. This component is used when a list-of-value button is pressed (see Entry_Field above). It displays all legal values for a field and optionally allows one to be selected. A sample representation is shown in Figure 26.
- Help_Message. The Help_Message is a window which appears when a help button is pressed. It displays information pertinent to the area the user is working in.

5.2.2.3 Identification of Object Relationships. Figure 27 shows how the objects in a form are related to each other. A form consists of one Form_Window. The Form_Window has one Action_Buttons component. It may have one Scroll_Area. A form may have zero or more Work_Buttons, Labels, or Entry_Fields (no entry fields would occur on a display-only form). Each entry field may have an optional help or list button associated with it. Each of these will have a Help_Message and Values_List associated with them respectively.

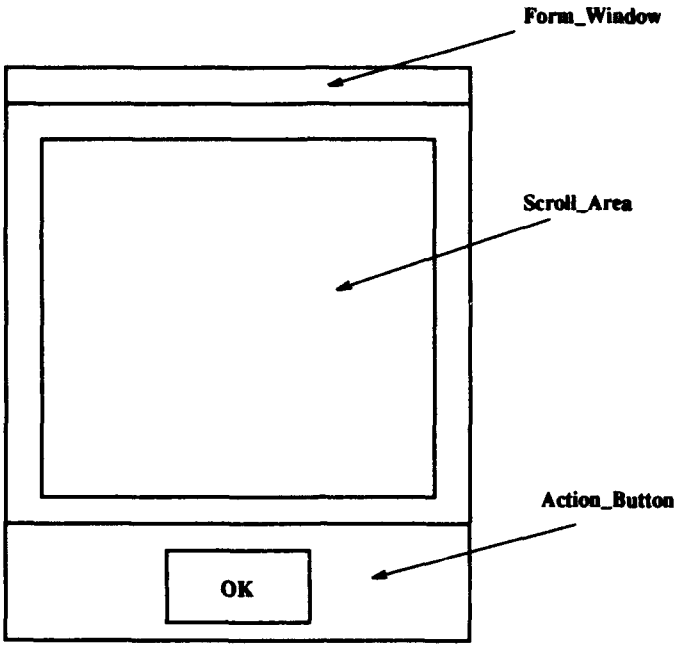

Figure 26. Input Form for List of Values

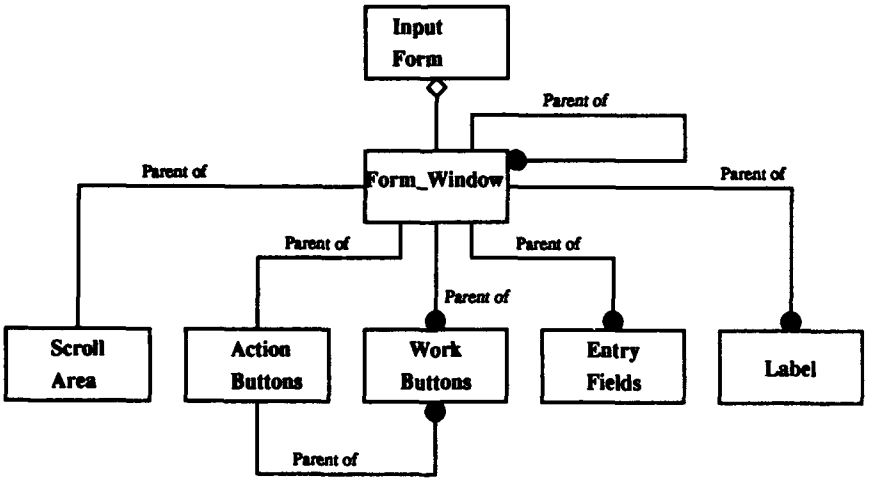

Figure 27. Mission Entry Form Object Model

Note that there are aggregations of objects. Action_Buttons is actually a grouping of several Work_Buttons. Values_List is actually a read-only form that consists of a Form_Window, a Scroll_Area, and a Work_Button. Help_Message is similarly defined. It is useful to think of these as simple components, however, since they are used so often as a grouping.

5.2.2.4 Identification of Object Methods. This section defines the methods each object requires to operate. All objects are assumed to have constructors (creation methods), destructors (deletion methods), and attribute access methods.

- Form_Window. This object class requires only the base constructor and destructor methods.
- Action_Buttons. This class requires a method to signal activation.
- Scroll_Area. This class has methods to scroll the display area up, scroll the area down, and clear it.
- Entry_Field. This class has a method for validation after the user enters a value. An implicit method for data entry is assumed.
- Work_Button. Like the Action_Buttons, this object class has a method to signal activation.
- Label. This object class has only the base methods.
- Values_List. Like the Scroll_Area, this class has methods to scroll the display area up and down. It also has a select method for when a user picks a value from the list.
- Help_Message. This class has only the base methods.

5.2.3 Integration with Simulation Engine. The user interface required that a menu item be added that allowed the "White" player (the game controller) to start the simulation without exiting the user interface. With this in place, it would appear to the user that the simulation was an integral part of the system.

5.3 X Window Binding Conversion Strategies

Recognizing that the outdated STARS bindings were a serious liability, Moore had earlier attempted to convert Saber (14).

With the SERC bindings based on the STARS bindings, converting a STARS application to a newer set would seem a simple task. Small applications do convert easily. Large applications, however, require a careful approach or excessive work will be done for a product that is less than optimal. This section discusses two strategies for conversion: low-level and high-level mapping.

5.3.1 Low-Level Mapping Strategy. Using the low-level style of conversion, the programmer determines either all at once, or incrementally, all the STARS functions, procedures, and types in an application. Each of these is then iteratively converted to a corresponding function/procedure/type in the target binding set.

5.3.1.1 Method. Previous work by Moore (14) describes a straight-forward method for converting each STARS identifier:

1. *Find Base STARS Declaration.* Identify the STARS source code statements that declared the identifier. This process may need to be nested, as the goal is to locate a standard Ada type definition.
2. *Find Similar Identifier.* Search the target binding specification for an identifier with the same or similar name as the STARS identifier.
3. *Find Base Target Binding Declaration.* Identify in the target binding how this identifier is declared. This process may also need to be repeated to locate the base Ada type declaration.
4. *Convert Between Base Declarations.* Using the declarations from above, the base type of the STARS identifier is converted to an appropriate type used by the target binding identifier.

5.3.1.2 Example Conversion. Saber was originally coded using the STARS bindings. Figure 28 shows a piece of the code used to set the colors of terrain features.

```
procedure Initialize_Terrain_Colors( Political_Red     : XT.Pixel;
                                     Political_Blue    : XT.Pixel;
                                     Political_Neutral : XT.Pixel;
                                     .
                                     .
                                     .
                                     City              : XT.Pixel ) is

  begin
     Political_Red_Color     := Political_Red;
     Political_Blue_Color    := Political_Blue;
     Political_Neutral_Color := Political_Neutral;
     .
     .
     .
     City_Color              := City;
  end Initialize_Terrain_Colors;
```

Figure 28. Section of Saber's STARS code

```
procedure Initialize_Terrain_Colors( Political_Red     : X_Lib.Pixel;
                                     Political_Blue    : X_Lib.Pixel;
                                     Political_Neutral : X_Lib.Pixel;
                                     .
                                     .
                                     .
                                     City              : X_Lib.Pixel ) is

  begin
     Political_Red_Color     := Political_Red;
     Political_Blue_Color    := Political_Blue;
     Political_Neutral_Color := Political_Neutral;
     .
     .
     .
     City_Color              := City;
  end Initialize_Terrain_Colors;
```

Figure 29. Low-level conversion of code

In this example, the code will be converted to the Ada/Motif binding set using low-level mapping.

1. *Find Base STARS Declarations.* The only STARS identifier used in Figure 28 is `XT.Pixel` (a type identifier). Looking in the STARS declarations `XT.Pixel` is defined as a subtype of `AFS_LARGE_NATURAL`. Looking in the boeing_afs package, `AFS_LARGE_NATURAL` is a subtype of `AFS_LARGE_INTEGER`. Finally, it is found that `AFS_LARGE_INTEGER` is defined as an Ada `INTEGER`. This is the base Ada type that is needed.
2. *Find Similar Identifier.* There is no `Pixel` in Ada/Motif's Xt Intrinsics library package, however there is one in the X library package.
3. *Find Base Target Binding Declaration.* Following a procedure similar to that above, `X_Lib.Pixel` will be have a base type of an integer range (`-2 ** 31 .. (2 ** 31) - 1`).
4. *Convert Between Base Declarations.* Since both `Pixels` are based upon integer values, the conversion is simply to substitute the new identifier for the old. Figure 29 shows the final product. Note that the package variables `Political_Red_Color`, `Political_Blue_Color`, etc. will also require this conversion in their declarations if the assignment is to compile properly.

   The converted code is shown in Figure 29.

5.3.1.3 Advantages. This type of approach works well with small applications having limited X functionality. It offers the following advantages:

- *Little knowledge of X required.* Since the method is somewhat algorithmic, a programmer with only a basic knowledge of X can do the simple comparisons needed.
- *No redesign required.* In essence, this is a line-by-line translation and avoids the need for extensive redesign of the application to fit a new binding.
- *Fast turn-around times.* Because it is a translation process, and because no design is required, the conversion can proceed rapidly.

5.3.1.4 Disadvantages. While tempting to use, this method fails on larger applications. As Moore (14) discovered, line-by-line translation quickly becomes unmanageable as more functions are introduced that interact with each other. In general, the disadvantages are categorized as follows:

- *Not robust.* A line-by-line translation is only effective if the number of translations are small, and they match up well with the target bindings. For example, this method fails when a STARS target function such as `Xt_Set_Widget` is encountered since there is no corresponding Xt identifier.
- *No optimization.* Since the STARS bindings were developed, X has added new functions that can significantly reduce the amount of code required. Since low-level mapping does not allow for redesign, these new features will not be used.
- *Prone to error.* Larger applications may have X values that are global in scope (such as default drawables); indiscriminate conversions of these values may introduce side-effects. A STARS binding may have a base type that maps to a similar, but incorrect target binding identifier. For example, `XT.Arg_List` in STARS appears to map to `Xt.Xt_Ancillary_Types.Arg_List_Ptr` in SERC (both are access types). If this mapping is used, the corresponding SERC functions using an argument list will not compile—they require `Xt.Xt_Ancillary_Types.Xt_Arg_List`.

5.3.2 High-Level Mapping Strategy. In larger applications, a more effective method of conversion will be high-level mapping. Using this approach the programmer takes the STARS application and breaks it down into sections of code that execute specific X tasks. These sections of code are then converted to the new set of bindings.

5.3.2.1 Method.

1. *Identify Sections of X Functionality.* Starting from the X application startup procedures, identify areas where specific tasks are being accomplished. For instance, an application may have a task that creates a form. This may have subtasks that create pushbutton and label widgets.

2. *Develop an Equivalent Set of X Calls.* For each section identified, develop an equivalent set of X calls. Care should be taken that the set of calls be extracted from a version of X that the new bindings are compatible with. The important idea here is to create the same functionality for the application using the best available X functions, procedures, and types. This may require redesigning some areas of the code.

3. *Map Equivalent Set to Target Bindings.* The next task is to map the X calls to Ada bindings. Commercially supported bindings such as Ada/Motif generally have supporting documentation that assists with this task.

4. *Convert Global Variables.* Care must taken that X global values which are used by different sections of code get converted. This will tend to happen naturally: as sections of code are converted, the global variables will become apparent.

These steps should be used as guidelines, and not be interpreted as strict procedure. Conversion from thin to thick bindings, for instance, may allow a relaxation on the equivalent set of X calls since there may not be a direct mapping.

5.3.2.2 Example Conversion. In this example, another section of Saber will be converted. This piece of code, shown in Figure 30, creates a label on the system's startup form.

1. *Identify Sections of X Functionality.* The code in Figure 30 is a good example of an X task. It creates a label with a string in it. There is no need for additional breakdown of the code.

2. *Develop Equivalent Set of X Calls.* An equivalent set of X calls to perform this function are:

```
XmStringCreateLtoR(text, charset)
XtSetArg(arg, resource_name, value)
XtCreateManagedWidget(name,
                      widget_class,
                      parent,
                      args,
                      num_args)
```

```
.
.
.
                        -- Create the TITLE
Label_Text := XM.Xm_String_Create_L_To_R(
                    "Welcome to the SABER Post-Processing System",
                    XM.Xm_STRING_DEFAULT_CHARSET );
XT.Xt_Make_Arg_List( SIZE => 3, ARGS => Args );
XT.Xt_Set_Arg( Args, XM.XmN_Width, 4000 );
XT.Xt_Set_Arg( Args, XM.XmN_Unit_Type, XM.Xm_1000TH_INCHES );
XT.Xt_Set_Arg( Args, XM.XmN_Label_String, Label_Text );
Title_Label := XM.Xm_Create_Label( Form_Widget, "Title", Args );
XT.Xt_Set_Widget( Child_List, Title_Label );
XT.Xt_Clear_Arg_List( Args );
XM.Xm_String_Free( Label_Text );
.
.
.
```

Figure 30. Section of Saber's STARS code

```
.
.
.
                  -- Create the TITLE
Label_Text := Xm_String_Create_L_To_R(
                    "Welcome to the SABER Post-Processing System",
                     Xm_String_Default_Charset );
Xt_Set_Arg( Args(1), Xm_N_Width, 4000 );
Xt_Set_Arg( Args(2), Xm_N_unit_Type, Xm1000th_Inches );
Xt_Set_Arg( Args(3), Xm_N_Label_String, Label_Text );
Title_Label := Xt_Create_Managed_Widget("Title",
                                        Xm_Label_Widget_Class,
                                        Form_Widget,
                                        Args(1 .. 3));
Xm_String_Free( Label_Text );
.
.
.
```

Figure 31. High-level conversion

`XmStringFree(string)`

Note that X provides a function (`XtVaCreateManagedWidget`) that eliminates the need for an argument list. Ada, however, cannot use this function since it has a variable argument parameter list. The Ada/Motif binding maps this function to `XtCreateManagedWidget` instead.

3. *Map Equivalent Set to Target Bindings.* Each function is mapped to an equivalent set of Ada/Motif bindings. Note that because Ada can constrain an array, the `num_args` parameter on `XtCreateManagedWidget` is not used.

4. *Convert Global Variables.* As the mapping takes place, the variables `Args` and `Form_Widget` will have to be converted. Rather than try a low-level mapping conversion, it is better to determine what `Xt_Create_Managed_Widget` expects them to be, and then change them to the appropriate type, in this case `Xt_Arg_List` and `Widget` respectively.

   The results of the conversion are shown in Figure 31.

5.3.2.3 Advantages. This method works well on applications of any size, and has several advantages over low-level mapping:

- *Robust.* Since the strategy develops a set of X calls and maps these to the new bindings, it doesn't fail when a non-mappable STARS function or identifier is found. In Figure 30, `Xt_Set_Widget` will not appear in the equivalent set or the final code.
- *Enhanced performance.* Some new functions in the X Window System (such as `CreateManagedWidget`) take the place of several older calls. Using these in Ada eliminates extra bindings and makes the code execute more efficiently.
- *Avoids type incompatibilities.* One of the problems in low-level mapping that high-level mapping avoids is mismatching argument types. Where it was possible to think `Xt.Xt_Ancillary_Types.Arg_List_Ptr` should be used where `XT.Arg_List` had appeared using low-level mapping, it would be immediately obvious that `Xt.Xt_Ancillary_Types.Xt_Arg_List` is used as a parameter type for all Ada/Motif set argument list functions.

5.3.2.4 Disadvantages. High-level mapping is inherently more complex than low-level mapping. While it handles many cases that low-level can't, there are trade-offs:

- *Requires expertise in X.* The process for doing this type of conversion is not strictly defined. Programmers will need experience with X to identify functional areas and the best set of equivalent calls to use.
- *Slower turn-around times.* Using the newer features may require design changes in many areas of the application. Changing the design, creating equivalent calls, and mapping them to bindings will require significantly more time than line-by-line translation.

5.4 Design of Mission Entry Forms

5.4.1 Design Objective. The main design objective for the mission entry forms was to ensure that the components used to build them could be reused. As noted in the analysis, each form consists of very similar elements. By properly defining these components as objects, it is possible to determine their state diagrams.

The form being developed from these objects operates as an event driven virtual machine. The events defined for the form are the events that can occur on any of its component objects. As events occur, the objects in the form execute predefined and developer defined methods. Pressing the Save button, for example, would invoke the Motif-defined code that changes the appearance of the button. This would then be followed by an activation of the developer defined database output routines. Note that the developer's output routine would have its own state diagram as it executes. Since the main interest is in visible components, this level of detail is abstracted away.

5.4.2 Component Design. Each object class has a state diagram that describes the messages it can receive, and how it will react to them. These diagrams are used in conjunction with the object class diagram to model how the system works. This model can then be translated to a specific computer language.

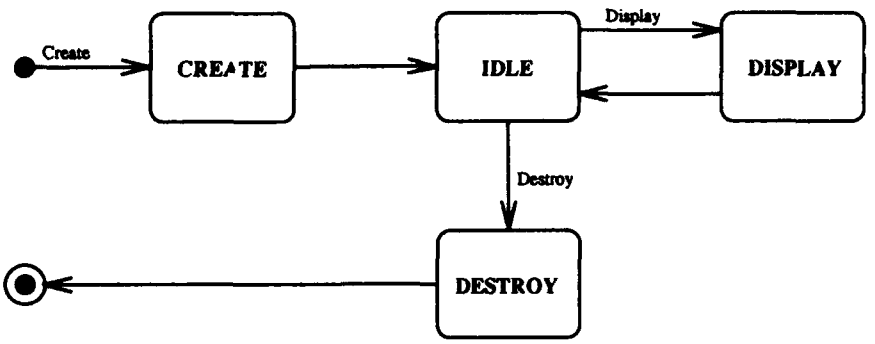

Figure 32. State Diagram for Form_Window Object

For the Saber user interface, the input form components have the following behavior defined:

5.4.2.1 Form_Window. Figure 32 is the state diagram for the Form_Window. The object receives a "Create" message and links itself into the X Window System. At this point other components can attach themselves to the Form_Window, but they will not be displayed. This prevents the annoying effect of seeing the X widgets drawn on the screen one-by-one.

When a "Display" message is received, the window will be managed by its X parent. This means that if the parent window of the Form_Window is displayed, then the Form_Window itself will be displayed. Any components attached to the window will also be displayed. If the parent is not displayed, the Form_Window will not appear.

A "Destroy" message causes the Form_Window to unlink itself from the X Window System. A "Destroy" message is sent to all of the components that attach to the window.

5.4.2.2 Action_Buttons. The Action_Buttons object is actually a grouping of several Work_Buttons (see Figure 27). The state diagram consists of the grouping receiving a "Create" message. It then sends corresponding "Create" messages to each of its Work_Buttons. The Work_Buttons then operate to gain the functionality of this object.

A "Destroy" message causes the component to send "Destroy" messages to each of its Work_Buttons. It then unlinks from X and frees any allocated system resources.

The state diagram for Action_Buttons is shown in Figure 33.

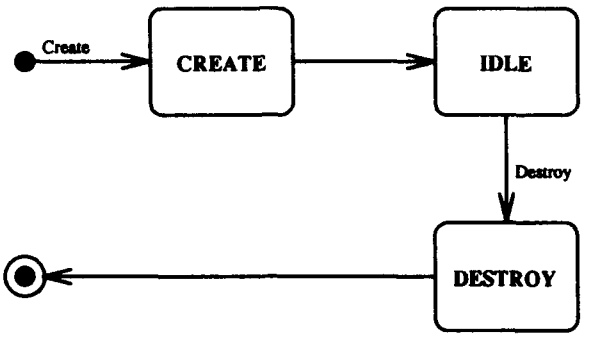

Figure 33. State Diagram for Action_Buttons Object

5.4.2.3 Scroll_Area. Figure 34 shows the state diagram for the Scroll_Area component. A "Create" message causes the scroll area to be created and initialized.

A "Select" message (caused by the user clicking on a displayed row) causes the scroll area to highlight the selected row. A "Select" message is sent to each Entry_Field on the form so they have the opportunity to change their displayed data to the newly selected information.

The "Unselect" message causes the scroll area to remove highlighting from any selected row.

A "Refresh" message causes all of the rows in the scroll area to be redisplayed. This causes changed data to appear properly, deleted rows to disappear, and new rows to appear.

A "Scroll_Up" or "Scroll_Down" message causes the scroll area to be shifted up or down respectively.

A "Bottom" message causes the scroll area to be shifted to the bottom of the list. This is used to display new tuples that have been added to the list.

The "Destroy" message causes the Scroll_Area to unlink from X and free any allocated resources.

5.4.2.4 Entry_Field. The Entry_Field is initialized when the "Create" message is received.

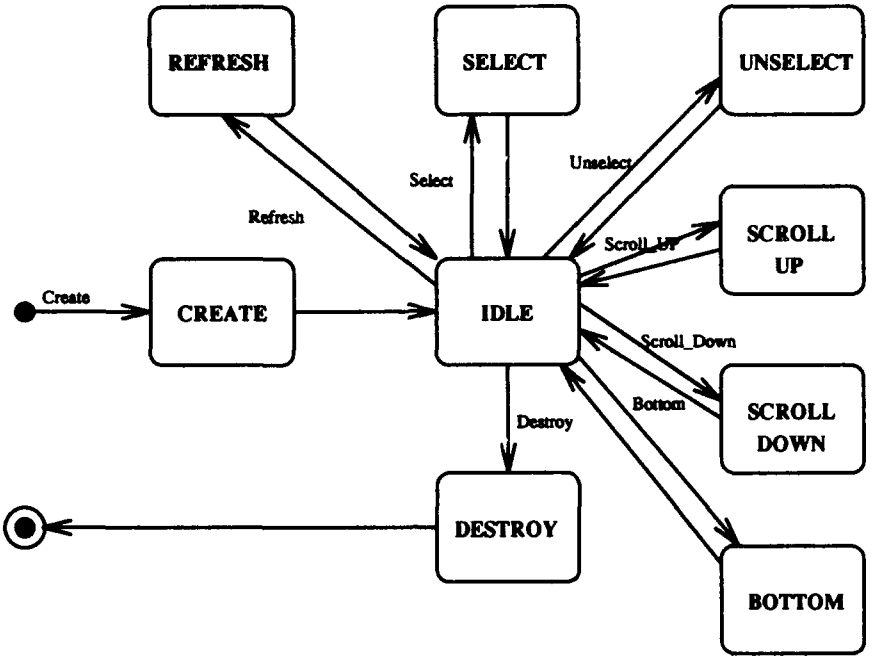

Figure 34. State Diagram for Scroll_Area Object

The "Activate" message is received when the user clicks on the field to begin editing its contents. When the user completes this editing, a "Validate" message is received. The Entry_Field then activates the procedure required to validate the data entered by the user.

The "Modify" message (sent by a Values_List) causes the field to replace the current value of its data with a value accompanying the message.

The "Destroy" message causes the Entry_Field to unlink from X and release any allocated resources.

The state diagram for Entry_Field is shown in Figure 35.

5.4.2.5 Work_Button. Like the other components, the Work_Button is created with a "Create" message.

When the user selects the button, an "Activate" message is sent to the button. The button then initiates the developer-defined action associated with it.

A "Destroy" message causes the button to unlink from X and deallocate system resources.

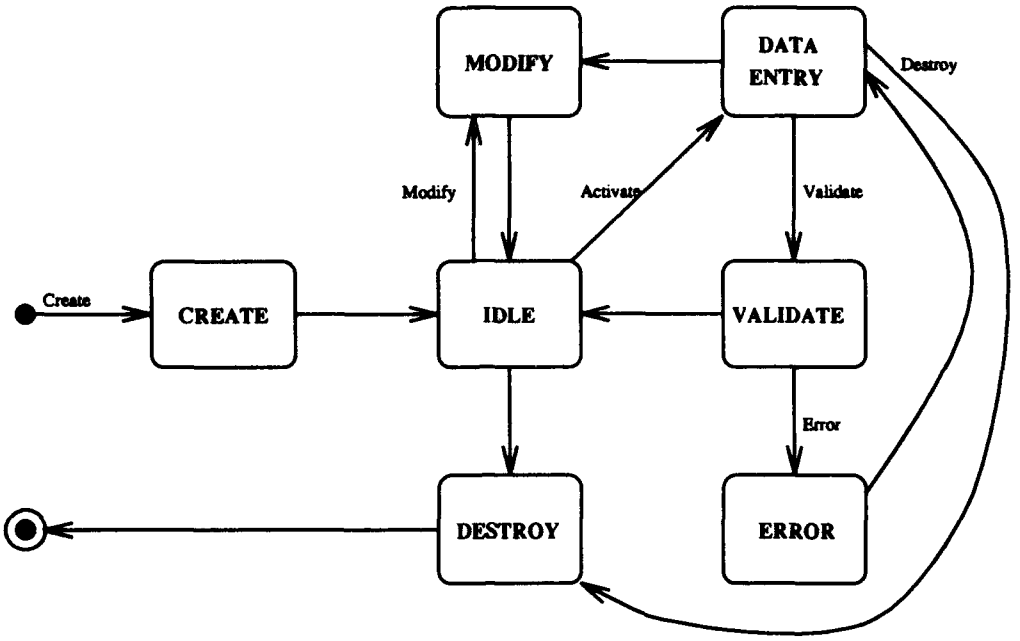

Figure 35. State Diagram for Entry_Field Object

The state diagram for Work_Button is shown in Figure 36.

5.4.2.6 Label. The Label component has the state diagram shown in Figure 37. The "Create" message creates the label.

A "Modify" message changes the value in the label.

The "Destroy" message causes the component to unlink from X and deallocate system resources.

5.4.2.7 Values_List. The values list is an input form which has been aggregated into a single object. Its behavior, therefore, is largely determined by the messages sent by its internal objects.

"Create" is issued by a Work_Button in the calling Form_Window. The list is created as a modal window, which means no other windows can receive input while it is active (this is true of all Form_Windows).

A "Select" message from the Scroll_Area of the Values_List causes the selected row identification to be saved by the component.

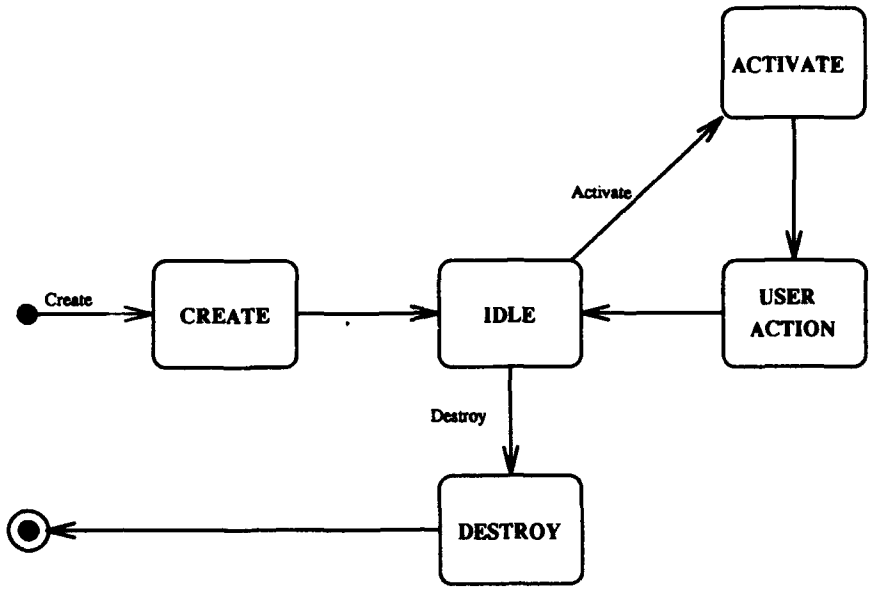

Figure 36. State Diagram for Work_Button Object

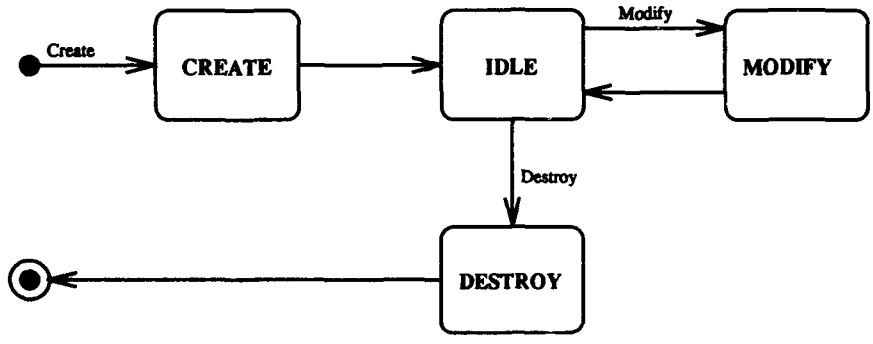

Figure 37. State Diagram for the Label Object

A "OK" message causes the selected row id (if any) to be sent to a specified Entry_Field as a "Modify" message. The Values_List then sends itself a "Destroy" message.

A "Cancel" message causes the Values_List to send itself a "Destroy" message.

5.4.2.8 Help_Message. Like the Values_List, Help_Message is an aggregation of objects. Its only messages are "Create" to create it, and "Done" which signals that the user wishes the window destroyed.

5.4.3 Land Movement Form. The land unit movement form has all of the components described above. The layout of these components was shown in Figure 23. Because an input form is event-driven, it really has no state diagram for the entire form. Rather, the state of the form is defined by the individual states of each form component.

To complete the design, we simply define the data upon which each component will operate, and specify what those operations are. The object class and state diagrams can then be combined with this to produce code.

5.4.3.1 Land_Unit_Movement_Form. This is a Form_Window upon which other components will be placed. It has no inherent attributes.

5.4.3.2 Land_Unit_Scroll_Area. This is a Scroll_Area which moves through the list of entered move orders. The basic internal attributes are the set of move order tuples currently being displayed and the identification of the order currently selected (if any). The scroll area must also be able to access a data store containing the rest of the orders. This supports its ability to scroll through the list.

5.4.3.3 Delete_Mission_Button. This is a Work_Button which deletes the movement order from the order list. It then sends a "Refresh" message to the scroll area.

5.4.3.4 Clear_Fields_Button. This Work_Button sends empty "Modify" messages to each Entry_Field so that they are cleared.

5.4.3.5 Add_Mission_Button. This is a Work_Button which, when selected, adds the entry composed in the data entry fields into the system. This is done by validating the fields and inserting them into the data store. An "Unselect" message is sent to the scroll area so that the currently highlighted row is un-highlighted. A "Bottom" message is then sent to the scroll area so that the new record is displayed.

5.4.3.6 Data Entry Fields. There are five Entry_Fields that are used to enter new records and edit old ones. All fields execute a validation routine when the user presses the Tab key, the Enter key, or presses the Add_Mission_Button.

5.4.3.7 OK_Button. This is an entry in the Action_Buttons. It causes the data store worked on during this dialog session to be made permanent in the database. A "Destroy" message is then sent to the Land_Unit_Movement_Form.

5.4.3.8 Cancel_Button. The opposite of the OK_Button, this button causes all work done during the dialog session to be aborted. A "Destroy" message is then sent to the Land_Unit_Movement_Form.

5.5 Summary

In this chapter we have examined two areas that are critical to eventual completion of the user interface: methods for converting out of the STARS bindings and the design of generic components for input forms. Two methods were examined for converting from STARS to newer sets of bindings: low-level mapping and high-level mapping. High-level mapping was found to be a superior technique. After examination of several mission entry forms, a set of common visual components were identified and designed using OMT. In the next chapter we discuss their implementation in Ada.

VI. User Interface Implementation

6.1 Overview

Applying the conversion process defined in the previous chapter to Saber yielded conversion rates much faster than anticipated. The high-level strategy was very successful, and this chapter discusses some of the reasons that contributed to this success. Some of the difficulties encountered while performing the conversion are also covered.

Development of the input form components required some translation to bring it from the object-oriented design in Chapter V to a non-object-oriented language like Ada83. We discuss how each component was built and present observations about combining the components to create the land unit movement form.

6.2 Interface Binding Conversion

6.2.1 Conversion Rate. Conversion of the interface occurred in two phases. First, during the development of the algorithms, the Game_Player and Main_Controller packages were converted for testing purposes. Consisting of over 2,000 lines of code, this process was completed after about 40 manhours of work.

In the second phase, after the algorithms had been fully developed, the rest of the simulation was converted. The remaining 21,000 lines of code were completed in three weeks (about 120 manhours of work). This appears to support the conclusion that high-level mapping is a viable strategy for large systems.

6.2.2 Common Code. One of the reasons that the high-level mapping strategy worked so well was the similarity between code segments. Figure 38 shows an example of a label created in the Terrain package. Figure 39 shows a label created in the Base package. Note that the only differences are the text of the labels and the name of the form they are associated with.

Between 70 and 80 percent of the X Window System code in Saber is for setting up the system to execute. Examining this setup code revealed that with only some minor name and organization changes, different segments are nearly identical. Using this similarity,

```
Label_Text := Xm_String_Create_L_To_R( "Hex Information Form",
                                       Xm_String_Default_Charset);
Xt_Set_Arg( Args(1), Xm_N_width, 3500 );
Xt_Set_Arg( Args(2), Xm_N_alignment, To_Xt_Arg_Val(Xm_Alignment_Center) );
Xt_Set_Arg( Args(3), Xm_N_unit_Type, FIX.Xm1000th_Inches );
Xt_Set_Arg( Args(4), Xm_N_label_String, Label_Text );
Title_Widget := Xt_Create_Managed_Widget( "Hex Title",
                                          Xm_Label_Widget_Class,
                                          Form,
                                          Args(1..4) );
```

Figure 38. Sample Terrain Label Code

```
Label_Text:= Xm_String_Create_L_To_R( "Base Status Form",
                                      Xm_String_Default_Charset);
Xt_Set_Arg( Args(1), Xm_N_width, 3500 );
Xt_Set_Arg( Args(2), Xm_N_alignment, To_Xt_Arg_Val(XmALIGNMENT_CENTER) );
Xt_Set_Arg( Args(3), Xm_N_unit_Type, FIX.Xm1000TH_INCHES );
Xt_Set_Arg( Args(4), Xm_N_label_String, Label_Text);
Title_Widget := Xt_Create_Managed_Widget("Status Title",
                                         Xm_Label_Widget_Class,
                                         Form,
                                         Args(1..4));
```

Figure 39. Sample Airbase Label Code

it was possible to use the editor to perform simple cut, paste, and global replacement functions that converted the code quickly.

This observation supports the idea that the input forms should consist of common components. With such minor differences between setup items, it naturally makes sense to encapsulate them into single segments of code that are used by the entire system. Thus, it should be possible to extend the input form components to include components for any wargame user interface.

6.2.3 Code Savings. One immediate benefit of converting the interface was a significant reduction in the amount of X related code. Each code segment dropped at least two lines of code: the statement that created an argument list, and the statement that freed it. In addition, most segments had separate calls to manage widgets. This was unnecessary with the Ada/Motif bindings. In general, there were 15 percent fewer binding calls in the converted code than the original.

6.2.4 Binding Errors. During the conversion process, two errors were discovered in the Ada/Motif bindings. The first involved how units of measurement are specified in an application. By default, the unit of measurement in Motif is the pixel. In general, it is not a good idea to accept this since X platforms will have varying sizes of monitors with varying resolutions. In some cases this can cause images to appear warped when shown on various systems.

The solution is to use a unit of length rather than the pixel. Saber uses the units specified in 1000th's of inches. In Motif, this is done by passing `XM_1000TH_INCHES` as a resource. Unfortunately, Ada/Motif specified this parameter incorrectly, and left out a needed conversion function. To correct this, it was necessary to create a special version of the identifier that had the correct values and put it in a package. Each user interface routine includes this package and uses the `XM_1000TH_INCHES` in it instead of the Ada/Motif version.

The second error was discovered while experimenting with faster drawing routines. The `Xt_Set_Clip_Mask` function is incorrectly bound to the C library. Instead of passing

(X,Y) for the mask coordinate, it passes (Y,Y). This was corrected by creating a new binding for the function and including it in the code as needed.

6.3 Mission Entry Components Implementation

Like the database interface described in Chapter IV, the components relied on the use of Ada packages for object attribute and method encapsulation. The following sections describe how this was accomplished.

6.3.1 Messages. Ada does not directly support passing messages between objects. In Saber, this is simulated by creating procedures and functions in each component's package that match the names of the message. The first parameter in these procedures is always an identifier of the object to be operated on. The other parameters correspond to information that would be contained in a message.

When an object is created it may be assigned a specific name. This name is represented as a string of 1 to 15 characters. It is also assigned an arbitrary object identification, a 32-bit integer value. A special object called a Resolver binds the name and class of an object to its 32-bit value. Using these identifiers and with knowledge of each object's type, messages can be sent to components in two ways:

1. *By Object Identifier.* If the specific object identifier is known, a message can be sent directly to the object. This is done by calling the appropriate message procedure in the target object class package. This procedure is then able to directly locate the object and operate on it.
2. *By Object Name.* If the object identifier is not known, but the object has been created with a specific name, this can be used for routing the messages. This is done by calling the appropriate message procedure in the target object class package. The name is then translated into a corresponding 32-bit id for the object. The procedure then operates on that object. Note that all message procedures are overloaded so that they can be called either with a name or a 32-bit value. This is useful when a

Motif callback executes and may not know the identifier of an object it needs to send a message.

6.3.2 Component Descriptions. Using Ada's package structure, it was a fairly simple matter to build the components. The package specification for each component detailed which messages would be received. Procedural implementations for acting on the messages were then written in the body. The following paragraphs briefly describe the unique characteristics of each component that was constructed.

6.3.2.1 Form_Window. The Form_Window package contains procedures for each of the messages it can receive. The actual implementation of the window is as a Motif `XmDialogShell` with a child `XmForm` to contain and manage the other components.

6.3.2.2 Action_Buttons. The Action_Buttons package has only two procedures, `Create` and `Destroy`. The `Create` call is passed an array that contains setup information for the Work_Buttons that will be controlled by this package. This setup information includes the size of the button, its label, and the address of the routine to be activated when the button is pressed.

6.3.2.3 Scroll_Area. This package is implemented as a Motif list widget. When created, a list of strings representing the current orders must be passed in. As entries are added and deleted, the list is automatically adjusted by Motif. Procedures are defined in the package for each of the messages it can receive.

6.3.2.4 Entry_Field. This package is implemented as a Motif text widget. It requires, routines to convert the field to a compound string, convert it from a compound string, and validate the field.

6.3.2.5 Work_Button. The Work_Button package implementation is a Motif pushbutton widget. At creation it is passed the size, position, and label of the button. In addition, the address of an activation routine is passed for when the button is pressed.

6.3.2.6 Label. The Label package is implemented as a Motif Label widget. At creation it is passed the size, label text, and position of the widget.

6.3.2.7 Values_List. The values list is a small input form created with the previously defined components. It requires the same parameters as the Scroll_Area, and the *name* of the entry field which should receive a "Modify" message containing the selected value.

6.3.2.8 Help_Message. This component was designed as a small input form. Motif, however, includes a message widget which accomplishes this function. The Help_Message implementation is accomplished using this rather than as a grouping of labels and a button. It must be passed an array of strings at creation that define the help text.

6.4 Land Unit Movement F- n Implementation

The form for land unit movement orders was implemented in a single package. The specification of this package has two procedures: Initialize and Activate. The Initialize routine creates the components needed for the form and defines the validation and conversion routines they require. The component's purpose and characteristics are discussed in the following sections.

6.4.1 Resolver. As described previously, the resolver package routes messages between the various components in the form. Each component created by the land unit movement form has its name and class bound to an object identifier by the Resolver.

6.4.2 Land_Unit_Movement_Form. This is the base window of the form. It is created from the Form_Window class. Its direct children are the Land_Unit_Scroll_Area, the OK, Cancel, Clear, Add, Delete buttons, and the data entry fields.

6.4.3 Land_Unit_Scroll_Area. This component is derived from the Scroll_Area class. It is created such that it can display the attributes of a movement order. This is retrieved and stored in the database as an IM_Land_Mission object. See Appendix B for a definition of this object.

6.4.4 Data Entry Fields. There are five data entry fields for the form: period, day, unit, mission, and target. Each entry field is associated with a validation routine. The period and day are checked for valid ranges. The unit is compared against the list of available units to the player. This mission must be a valid mission type: ATTACK, DEFEND, WITHDRAW, or MOVE. The target of a mission may be any valid hex, airbase, or unit.

6.4.5 Land_Unit_Action_Buttons. The action buttons are created by the Action_Buttons package. An array is passed that gives the label and an action routine address (a callback) for each button. The purpose of each of these buttons is as follows:

6.4.5.1 Delete_Button. Deletes the currently highlighted row in the Land_Unit_Scroll_Area. If no row is highlighted, an error message is generated. This error is displayed using a Help_Message object.

6.4.5.2 Clear_Button. Clears each of the data entry fields and removes highlighting from any selected row in the Land_Unit_Scroll_Area. This gives the user a clean slate for creating new orders.

6.4.5.3 Add_Button. Collects the data in the entry fields and creates a new order. If a row has been selected in the Land_Unit_Scroll_Area, then it is replaced with the new record. If now row is selected, then the record is inserted into the list at the correct day/period location.

6.4.5.4 OK_Button. The OK button signals that the orders are correct. The action routine makes the changes permanent in the database.

6.4.5.5 Cancel_Button. The Cancel button signals that the orders are not to be saved. The action routine removes the temporary orders from the database.

6.5 Summary

This chapter discussed the conversion of the Saber user interface to the SERC Ada/Motif bindings and the implementation of the mission input form components in Ada. Conversion using the high-level mapping method proved to be even faster than expected. This is attributed to the large amount of common code that exists in an X application. The mission input form components were implemented as separate packages. Each package acted as an object manager, allowing messages to be passed to its objects and invoking methods as needed. The land unit mission entry form was developed using these components.

VII. Research Analysis

7.1 Overview

The integration of Saber into a cohesive system involved the construction not only of Ada code, but the creation of new development methods and tools. In this chapter we examine the effectiveness of these methods and tools by analyzing the results obtained in the implementation chapters.

7.2 Analysis

In this section we examine the work performed in Chapters III and V compared with the implementation outcomes of Chapters IV and VI. This covers the database interface, the binding conversion strategies, the reusable X components, and some general observations about object-oriented programming with Ada83. It concludes with a discussion of the impact these areas had on the Saber integration.

7.2.1 Reusable Database Interface. Using Rumbaugh OMT method for analysis and design we created the database objects and Ada specifications before we actually determined that the SAIC OODBMS would be used. Once we corrected the bugs in the OODBMS, it was possible to fill in the bodies of the packages.

While constructing the bodies of the packages, we noticed that many of the class methods were very similar. After constructing 12,000 lines of implementation code, we determined that a generic package could be used instead. The generic package eliminated the need for 10,000 lines of code—a tremendous savings. It is especially significant, however, in that no changes were required to the application interface. Code written to test against the non-generic interface still worked when compiled with the generic interface. This substantiates the usefulness of object-oriented development using the OMT method.

In the following sections we discuss some of the advantages and disadvantages of using the interface and the SAIC OODBMS.

7.2.1.1 Flexibility. The underlying generic body code was re-written several times to improve performance. At no time did these changes mandate alteration of the

application interface. Even those which radically changed the storage method were well insulated from the application.

The price paid for this type of flexibility is that the interface tends to appear more as an object server than as a full-fledged OODBMS. It lacks OODBMS functionality in several areas: dynamic queries, version control, sophisticated locking mechanisms, etc. These are sacrificed so that any database can be hooked into the interface. For the purposes of Saber, however, the interface is more than adequate.

7.2.1.2 OODBMS Performance. The SAIC OODBMS performance was adequate for the random access of objects, but was slower than expected when performing bulk reads and writes. This was especially apparent in the attempts to load the hexagon map into the user interface.

One cause of this slow performance may be in the large overhead imposed by the code structure. Profiling the map reading operation of the simulation revealed that the client spends 40 percent of its time processing and 60 percent of its time waiting for the server. Since the client is merely requesting data from the server, the expected processing time should have been more on the order of 10 or 20 percent. Analysis of the profile shows that there are numerous database system routines that use 1 to 4 percent of the processing time. This overhead may be the result of nested generics, a potential problem noted by Arnold Voketaitis in his development of an RDBMS interface (26:66). The profile is shown in Table 1. The field *% Time* represents the percentage of total execution time the module used. The *Time* field represents the amount of time the program has been executing at the end of that module.

Another potential problem is the data overhead attached to each object. Analysis of the server revealed that creation of an object is done by repeatedly calling the parent object class' creation routine. For some types of objects, particularly binary large objects, this adds a lot of extra information that must be written to the database. The general effect observed with the hex object was that four times as much information was written than the object actually contained. Taking I/O block reads into account, this results in twice as many database reads as necessary.

Table 1. Partial Listing of Client Map Read Profile

% Time	Time	Procedure Name
58.4	3.59	_sigtramp
4.3	3.85	system_v_semaphore.operate
4.1	4.10	allocation.is_null
2.8	4.27	communications_buffer.allocate_string
2.0	4.39	database_interface.send
2.0	4.52	_memcpy
1.6	4.62	_semctl
1.6	4.72	allocation.allocate_the_space
1.6	4.82	fat_heap_global.aa_global_new
1.5	4.91	allocation.get_allocation_info
1.5	5.00	fat_heap_global.aa_global_free
1.3	5.08	communications_buffer.allocate_string
1.3	5.16	dbms_interprocess_control.release
1.3	5.24	dbms_interprocess_control.wait_for
1.2	5.31	communications_buffer.allocate_object_id
1.0	5.37	allocation.length
0.8	5.42	allocation.report_change
0.8	5.47	communications_buffer.allocate_object_id
0.8	5.53	BCOPY
0.8	5.58	database_interface.send_message
0.7	5.62	system_v_semaphore.acquire

7.2.2 Binding Conversion Strategies. The high-level mapping strategy proved to be very effective in converting the original Saber user interface. Over 23,000 lines of code were converted with approximately 160 manhours of work. This fast conversion was attributed to the fact that 70 to 80 percent of X application code is used for setup. This code has similar characteristics which allows for easy use of cut, paste, and global replacement techniques.

Since the converted code used the newer version of Motif, it was able to use the more efficient library calls. This resulted in a 15 percent reduction in the number of binding calls.

7.2.3 Reusable X Components. The X components developed worked extremely well and appear to have application outside the domain of the input forms. After writing the initial packages to implement the components it was a simple matter to connect them into a working input form. There had been some concern that the overhead of passing messages might be intrusive. Since the interface is only passing a few messages per minute, this turned out not to be a factor.

7.2.4 Use of Ada83. Ada is a well-designed language that more than proved its worth during the development of this project. Generics eliminated thousands of lines of code. Strong-types and constraints made the development of X code much safer than equivalent development in C. The package structure made it possible to easily translate the detailed designs into implemented code. These areas are discussed in the following sections.

7.2.4.1 Generics. Generic packages were used throughout the code developed for this thesis. They provided a convenient mechanism for simulating inheritance from parent objects. Unfortunately, none of the object-oriented design notations take into account the idea of a generic template. This sometimes made it difficult to represent the class structure.

7.2.4.2 Motif Programming. One of the major problems encountered while writing some test Motif programs in C was the ease with which incorrect parameters could be passed to library routines. Using the Ada/Motif bindings, all parameters were checked to ensure they were of the correct type at compile time. Constraint errors caught those cases where incorrect values were passed at run-time. In general, Ada provided a safe environment for constructing the user interface.

7.2.4.3 Object-Oriented Programming. One problem that required considerable work-around was the lack of object-oriented constructs in Ada83. Ada83 supports some basic object concepts (via the package structure), but it lacks inheritance and dynamic dispatch of objects. To work around this procedures and functions were created that had the same names as the messages. The parameters specified which object the message was for, and the procedure then operated on it. While this made the syntax rather awkward, it did not impose serious overhead.

7.2.5 Ada9X Issues. It is anticipated that by the end of 1994 the proposed new Ada language standard will have been approved. This new standard will add the necessary constructs for the language to directly support object-oriented concepts missing in Ada83. A re-implementation of the abstract database design in Ada9X (where X is the year of approval) would significantly change the impact of persistent types on existing simulation code. In this section we discuss this impact.

7.2.5.1 Persistent Types. Ada9X provides the key mechanism needed to make persistence a possibility: the *Adjust* operation. It allows the user to define what action will take place at the time of an assignment. To use this operation, a record must inherit the Finalization.Controlled package supplied with the compiler. This same package also supplies two other functions that are useful: *Initialize* and *Finalize*. These functions allow the user to define how a variable is initialized when it comes into scope, and what happens when a variable goes out of scope.

We can use this functionality to implement a generic persistence package. A suggested package is shown in Figure 40. The developer supplies a type that they wish to

```
with Finalization;

generic

  type Persistent_Type is private;

package Persistence is

  type Controlled_Type is new Finalization.Controlled with
    record
      The_Type : Persistent_Type;
    end record;

  -- This procedure overrides Initialize in the Finalization package
  procedure Initialize(Object : in out Controlled_Type);

  -- These two procedures define implementations for the abstract
  -- procedures in the Finalization package
  procedure Adjust(Object : in out Controlled_Type);
  procedure Finalize(Object : in out Controlled_Type);

end Persistence;
```

Figure 40. An Ada9X Package Specification for Persistent Types

make persistent. The package instantiates a version of this type that inherits from Finalization.Controlled. The Initialize, Adjust, and Finalize procedures are defined for this new type. This allows the generic body to be implemented such that it supplies the database implementation of these routines. The developer then declares variables by using the Controlled_Type defined within the package. A sample declaration is shown in Figure 41.

This method of persistence does not supply complete transparency. Note that the declaration of Core_Type is a record. Aggregate assignments between records will result in the Adjust procedure being called. Assignments between record components, however, will not result in Adjust being called (see Figure 42). Thus, the developer will still have to be aware of when objects will and will not be saved.

7.2.5.2 Persistent Objects. Ada9X may ease the programming requirements needed to implement persistent objects since the object-oriented constructs will be built

```
type Core_Type is
  record
    Name : STRING(1..20);
    Age : INTEGER;
  end record;

package Persistent_Core_Package is new Persistence(Core_Type);

subtype Persistent_Core_Type is Persistent_Core_Package.Controlled_Type;

Persistent_Core : Persistent_Core_Type;
```

Figure 41. A Sample Declaration of a Persistent Type

```
Core_One : Persistent_Core_Type;
Core_Two : Persistent_Core_Type;

begin
  -- These assignments are NOT stored since they will not activate
  -- the Adjust procedure
  Core_One.The_Type.Name := "John Smith        ";
  Core_One.The_Type.Age  := 21;

  -- This assignment to Core_Two is stored since it fits the Adjust profile in
  -- the instantiated package Persistent_Core_Package
  Core_Two := Core_One;

  -- Note that if we put a store function in the finalize procedure,
  -- Core_One WILL be stored at this point.
end;
```

Figure 42. Sample Persistent Assignments

into the language. Other than this, however, little change can be expected since the basic purpose is to have tight control over when an object is loaded from and saved to the database. Since these will still be basic Get and Put procedure calls, the introduction of object-oriented constructs will have no real impact.

7.2.6 Impact on Saber. Each of these areas had a major impact on Saber. In the following sections the code impact, maintainability aspects, and performance effects are discussed.

7.2.6.1 Code. Introduction of the database interface significantly reduced the amount of code needed to load data in both the simulation and user interface. On average, a database input or output routine required 30 to 40 percent fewer Ada statements than the flat-file routine. In the simulation, however, the flat-file input routines were kept so that they could be invoked to build the OODBMS from text files. This resulted in a net increase of 1,800 lines of code. Approximately 2,000 lines of code, however, were eliminated from the user interface.

The database interface consists of 9,700 lines of code. Most of these are contained in the Saber application object packages. The actual database consists of 40,000 lines of code, with 7,500 lines contained in some support libraries (linked-lists, hash tables, etc.).

7.2.6.2 Maintainability. Integration with the database interface has solved the major maintenance problem associated with the original Saber: consistent data formats. Any changes to the database will now be automatically reflected in both the user interface and the simulation. Its existence prevents one subsystem from unilaterally changing the data structure. This eliminates the possibility of crashing the other subsystem due to data format changes.

Utilization of the SERC Ada/Motif bindings contributed greatly to the future maintenance of the user interface. As Motif continues to be improved, the bindings will not become obsolete.

7.2.6.3 Performance. In general, it appears ill-advised to simply replace an RDBMS or flat-file system with an OODBMS. In the case of Saber, it was found that replacing the inefficient flat-file routines with the OODBMS improved performance. Had the flat-file routines been properly written, however, they probably would have been much faster than the corresponding OODBMS code. The user interface had severe performance difficulties with front-loading the hexagon map and had to be reverted back to reading the map from a file.

Loading the entire database into memory at the start of each subsystem's execution places a burden on system resources and causes the database to read data that may not be used. Judicious use of persistent objects would be a much better solution. For example, the simulation uses only a small fraction of the hexagon map. Rather than load all 10,000 hexes, only about 300 would be loaded during a simulation run. For those object classes that are used often, they could be locked into memory after a load occurs. This would act as a cache of sorts, preventing multiple reads from the disk.

7.3 Summary

In this chapter we analyzed the results of the implementation chapters. The user interface appears to be flexible enough for Saber's requirements, but is lacking full OODBMS functionality. The performance of the database was slower than expected; this may be attributable to database overhead. Ada provided excellent protection against common X programming errors, but required some work-arounds to implement the object-oriented design.

VIII. Summary and Recommendations

8.1 Overview

This final chapter summarizes what has been accomplished by the thesis. Saber still has many areas that require completion. Recommendations for how they should be approached are discussed. The chapter concludes with some final comments on this project.

8.2 Summary of Research

This thesis has covered a number of different areas: object-oriented programming in Ada, object-oriented database interfaces, X Window System programming in Ada, and the development of common X interface objects. In summary, we accomplished the following:

1. *Developed a Reusable OODBMS Interface.* A high-level reusable OODBMS interface was developed in Ada. It hides the implementation of the database from the application, allowing any OODBMS to be used with the interface.
2. *Integrated Saber Subsystems.* Using the database interface, the Saber simulation engine and user interface were integrated. A common set of database classes were defined and implemented. The subsystems are able to access the database without being aware of the physical data format of a database record. This has removed the tight coupling between subsystems in the original code. This resulted in a savings of 2,000 lines of code in the user interface. There was a net gain of 1,800 lines of code in simulation engine since the original flat-file routines were kept intact so that the OODBMS could be initialized from text files.
3. *Developed X Binding Conversion Strategy.* The Saber user interface was successfully converted to the SERC Ada/Motif bindings using a high-level mapping conversion strategy. This strategy emphasizes the idea of determining code functionality before attempting a mapping into a new binding set.
4. *Created Reusable X Components.* The mission entry forms in Saber were broken down into individual components that were modeled as objects. They were then implemented as Ada packages. This allowed any input form in Saber to be constructed

simply by connecting components and supplying input validation routines. The land unit mission entry form was constructed using this technique.

8.3 Recommendations

Saber has been an outstanding vehicle for experimenting with new ideas and techniques. Future work should expand on what has been done here:

- *Air Battle Completion.* Due to time constraints, it was not possible to add the components to the simulation and the user interface that perform air combat. Clearly defined in previous theses, these are merely an exercise in writing Ada code to implement them.
- *Continued Graphics Enhancement.* Another victim of time was the display of combat results. Previous efforts attempted to accomplish this using animation (10, 14). The X Window System is not re-entrant, however, and the tasks used to animate often crashed the system. A good method of displaying combat results needs to be developed and implemented in Saber.
- *Persistent Objects and Types.* As noted previously, the flat-file simulation had minimal impact on application code, but suffered in the area of performance. The simulation and user interface should be modified to use persistent objects and types as appropriate.

8.4 Final Remarks

This thesis has developed methods and tools that allowed the integration of the Saber wargame. Its impact, however, steps beyond just this wargame. The database interface provides a reusable tool to those who need simple access to an OODBMS. It stands as a good starting point for those wishing to develop more complex, yet equally portable, interfaces. The high-level mapping strategy for converting between bindings is applicable to any X application written in Ada. The generalized components developed for the user interface were also implemented as portable objects. Thus, any Ada application can use them to create visual interfaces. Saber's eventual completion will have a direct, positive,

impact on the Air Force as a valuable educational tool. The spin-off benefits of developing Saber, however, will have a much broader and greater impact.

Appendix A. Object Class Diagrams for Saber

A.1 Overview

This appendix contains the object class model for the Saber simulation. This model is represented as a series of Rumbaugh-style diagrams (20). They were constructed by comparing the diagrams and descriptions of Douglass (6) with the actual source code of the simulation and user interface. In most cases, the diagram contents are unchanged, but the presentation of the diagram has been altered for readability purposes.

Figure A.2 contains a new weapon type, biological. The source code often mixes the terms for chemical and biological weapons. It appears that there may have been intent to model both, but they were lumped into a single "Chemical" category. A finished version of the Saber simulation would actually split them into appropriate categories, so they have been shown separately in the diagram.

A.2 Diagrams

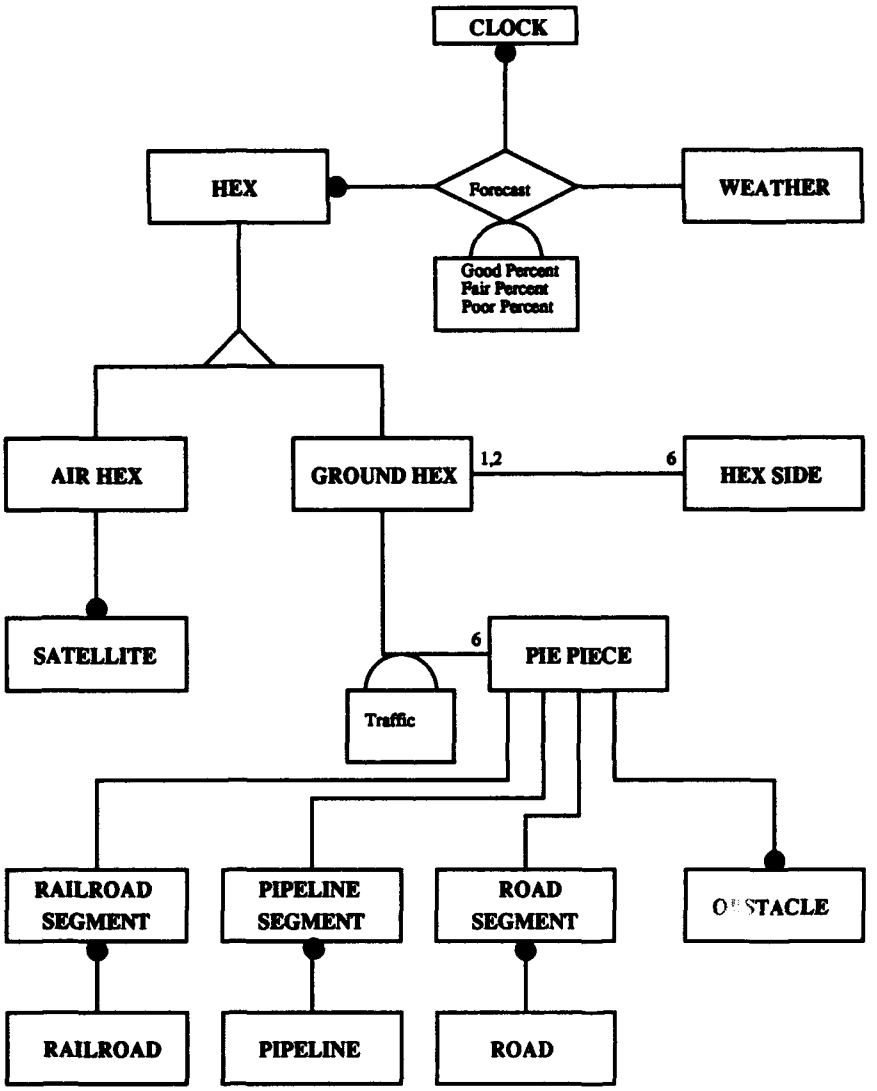

Figure 43. Map Hexagons and Features Organization

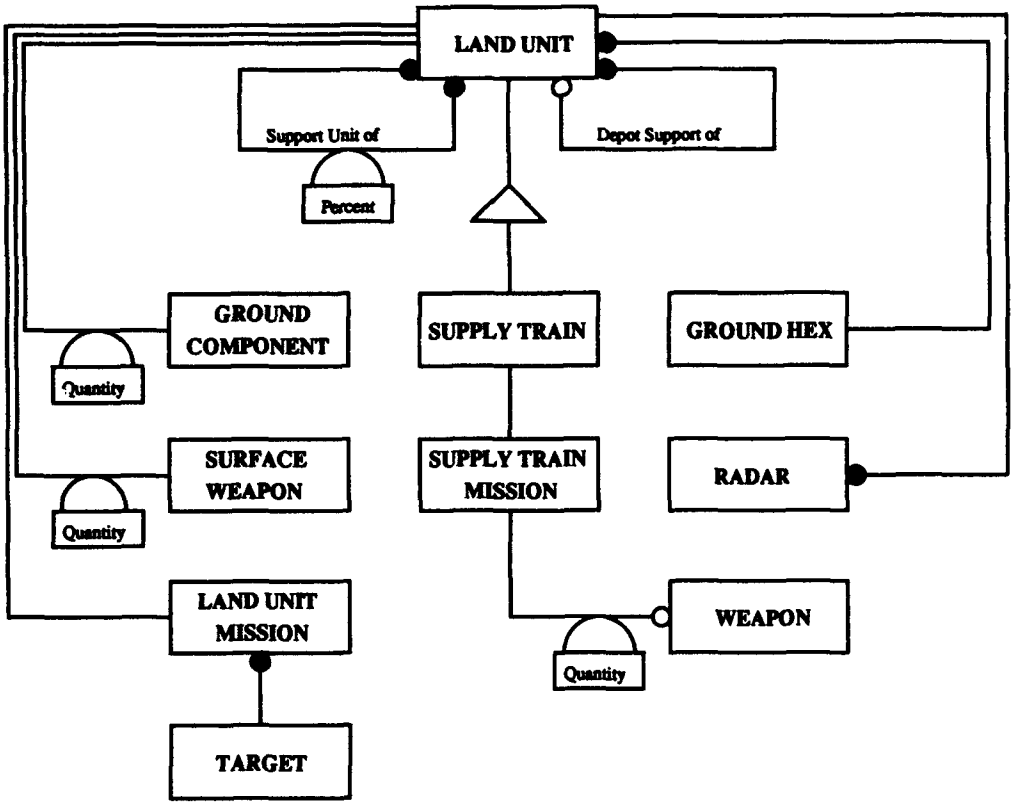

Figure 44. Land Unit and Components Organization

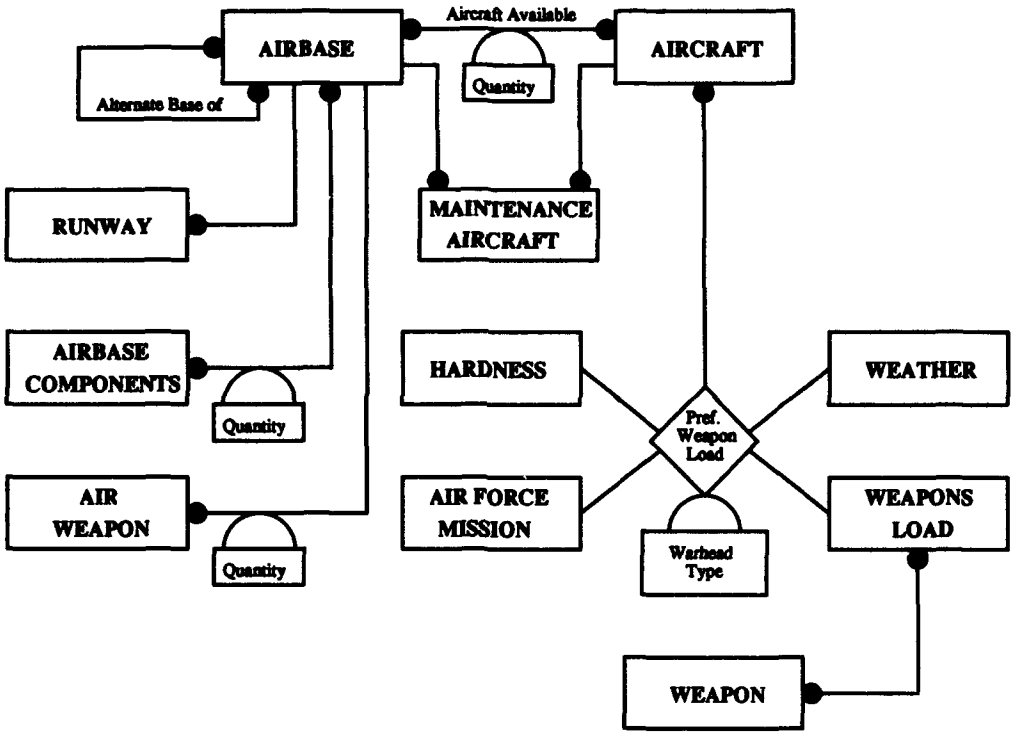

Figure 45. Airbase and Aircraft Organization

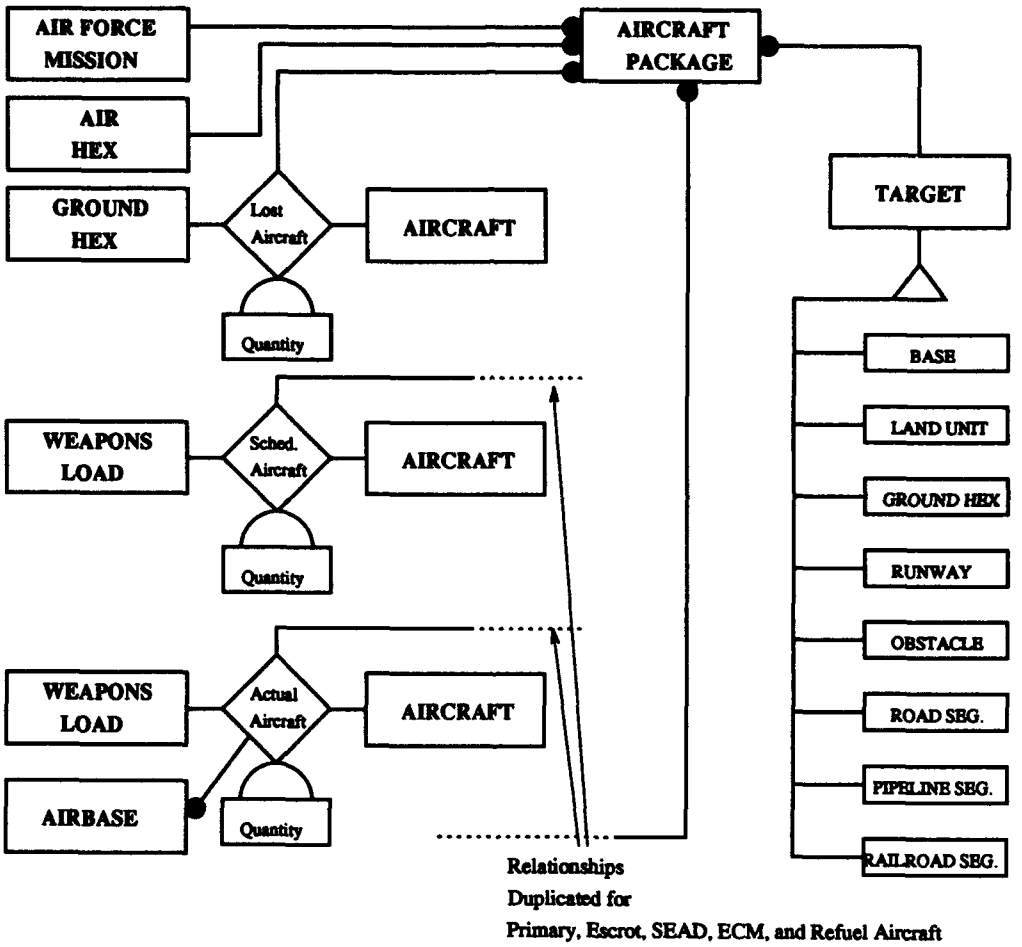

Figure 46. Aircraft Package Organization

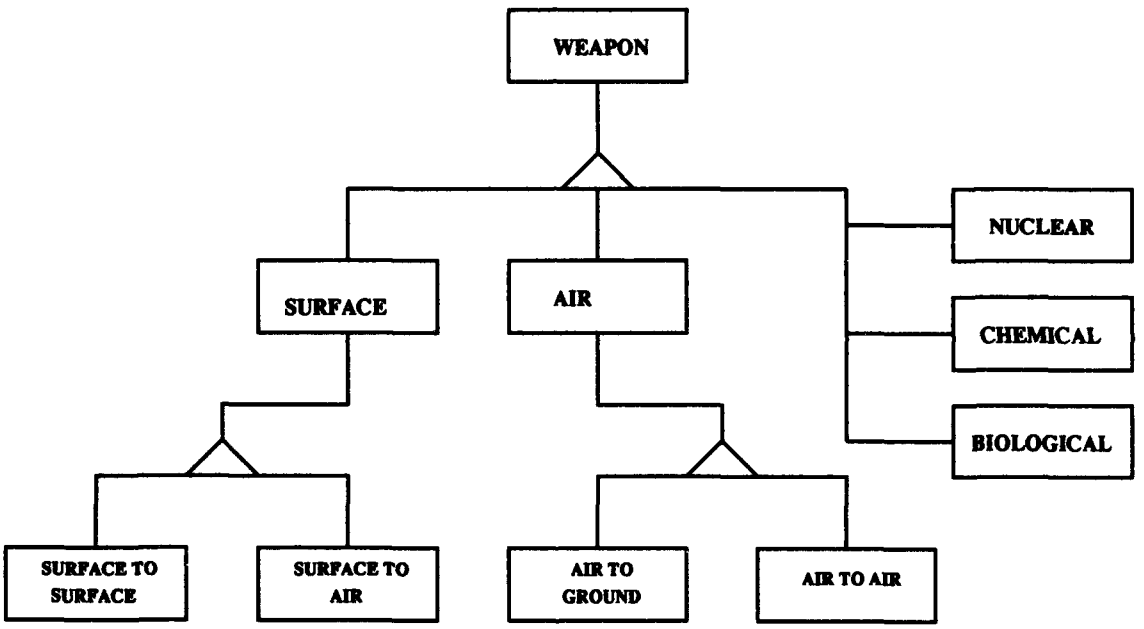

Figure 47. Weapon Organization

Appendix B. Database Object Definitions

B.1 Overview

Each simulation object in Saber has a corresponding database object. This appendix defines each of the database object classes that were created to support the Saber simulation and user interface. The attributes and object database methods that operate on them are listed in concise tables.

Each section is split into two parts—the attributes and the database methods. The attribute listing gives the name of each stored attribute and the Ada representation of that attribute. Some attributes are actually enumerated types that are stored as integers but have special semantic meaning to the program. The representation for these types is listed simply as "enumerated" since most have many possible values. The values can be found by looking in the source code or reading the descriptions in Douglass (6) or Horton (9).

The second listing consists of the database methods that operate on the objects. The "Key?" field of the listing refers to whether a method uses a key value or not. This will be YES or NO for methods that can use keys, or blank for those that cannot. If a method is used to associate two objects, then the target object class is also listed.

B.2 Definitions

B.2.1 IM_AA_Weapon. Contains attributes needed to represent an air-to-air weapon.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of weapon
Force	Enumeration	Country of origin
Miss_Range	INTEGER	Range of weapon
SSPK	FLOAT	Probability of kill

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.2 IM_AG_Weapon. Contains attributes needed to represent an air-to-ground weapon.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of weapon
Force	Enumeration	Country of origin
Lethality_Radius	INTEGER	Radius of destruction
CEP	FLOAT	Quality of weapon
Pk_Hard_Point_Type	FLOAT	Prob. of Kill for Hard Target
Pk_Med_Point_Type	FLOAT	Prob. of Kill for Medium Target
Pk_Soft_Point_Type	FLOAT	Prob. of Kill for Soft Target

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self

Method	Key?	Target Class
Get		Self
Put		Self
Delete		Self

B.2.3 IM_Air_Component_Link. Links an airbase object to a airbase component object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of component
Quantity	INTEGER	Number linked to airbase

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Unit		IM_Land_Unit
Get_Unit		IM_Land_Unit
Add_Component		IM_Air_Component
Get_Component		IM_Air_Component

B.2.4 IM_Air_Hex. Contains attributes needed to represent an air hex on the map board.

Attributes.

Attribute Name	Representation	Description
Air_Hex_Weather	Enumeration	Weather in this hex
Weather_Zone	INTEGER	Used for forecasts
Attrition	FLOAT	Attrition rate
EC	INTEGER	
Trafficability	Enumeration	Difficulty of travel
Persistence	INTEGER	Persistence of NBC weapons

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Blue_Acpkg		IM_Aircraft_Package
Remove_Blue_Acpkg		IM_Aircraft_Package
First_Blue_Acpkg		IM_Aircraft_Package
Next_Blue_Acpkg		IM_Aircraft_Package
Add_Red_Acpkg		IM_Aircraft_Package
Remove_Red_Acpkg		IM_Aircraft_Package
First_Red_Acpkg		IM_Aircraft_Package
Next_Red_Acpkg		IM_Aircraft_Package
Add_Satellite		IM_Aircraft_Package
Remove_Satellite		IM_Aircraft_Package
First_Satellite		IM_Aircraft_Package

Method	Key?	Target Class
Next_Satellite		IM_Aircraft_Package

B.2.5 IM_Air_Weapon_Link. Links an airbase object to an air weapon object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of component
Quantity	INTEGER	Number linked to airbase

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Airbase		IM_Airbase
Get_Airbase		IM_Airbase
Add_Air_Weapon		IM_Air_Weapon
Get_Air_Weapon		IM_Air_Weapon

B.2.6 IM_Airbase. Contains attributes needed to represent an airbase.

Attributes.

Attribute Name	Representation	Description
Base_No	NATURAL	Unique base id number
Full_Designator	STRING(1..30)	Name of base
Abbrev_Designator	STRING(1..8)	Abbrevriated version of base name

Attribute Name	Representation	Description
Command	STRING(1..15)	Base MAJCOM or equivalent
Country	STRING(1..4)	Country of origin
Force	Enumeration	Country of origin
HQ	STRING(1..5)	Headquarters of airbase
Move_Allowed	BOOLEAN	Can base be moved?
Pres_Loc	INTEGER	Present location of base
Fut_Loc	INTEGER	Future location of base
Mission	Enumeration	Base's current mission
Width	INTEGER	Largest width of runway
Length	INTEGER	Largest length of runway
Region	INTEGER	
Weather_Min	Enumeration	Worst weather value
Is_Base_Overrun	BOOLEAN	Has base been overrun?
Is_Base_Within_Enemy_Art	BOOLEAN	Can base be mortared?
Is_Base_Under_Nuc_Chem_Atk	BOOLEAN	Is base under NBC attack?
Is_Base_Under_Air_Atk	BOOLEAN	Is base under air attack?
Active_Enemy_Mines	INTEGER	Level of enemy mining
MOPP_Posutre	INTEGER(0..4)	
POL_Soft_Store	INTEGER	Petrol/Oil/Lubricant storage
POL_Hard_Store	INTEGER	Petrol/Oil/Lubricant storage
Max_POL_Soft	INTEGER	Petrol/Oil/Lubricant storage
Max_POL_Hard	INTEGER	Petrol/Oil/Lubricant storage
Maint_Pers_On_Hand	INTEGER	Maintenance personnel
Maint_Hrs_Accum	INTEGER	Accumulated maintenance hours
Maint_Equip_On_Hand	INTEGER	Maint. equipment available
Spare_Parts	INTEGER	Spares on hand
Max_Ramp_Space	INTEGER	Max parked aircraft
Ramp_Avail	INTEGER	Currently available ramp space

Attribute Name	Representation	Description
Shelters	INTEGER	Aircraft shelters available
EOD_Crews	INTEGER	
RRR_Crews	INTEGER	
Vis_To_Enemy	STRING(1..8)	Indicates visibility to enemy
Intel_Index	FLOAT	

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Runway		IM_Runway
Remove_Runway		IM_Runway
First_Runway		IM_Runway
Next_Runway		IM_Runway
Add_Air_Weapon		IM_Air_Weapon_Link
Remove_Air_Weapon		IM_Air_Weapon_Link
First_Air_Weapon		IM_Air_Weapon_Link
Next_Air_Weapon		IM_Air_Weapon_Link
Add_Alt_Base		IM_Airbase
Remove_Alt_Base		IM_Airbase
First_Alt_Base		IM_Airbase
Next_Alt_Base		IM_Airbase
Add_Aircraft_Type		IM_Aircraft_Link

Method	Key?	Target Class
Remove_Aircraft_Type		IM_Aircraft_Link
First_Aircraft_Type		IM_Aircraft_Link
Next_Aircraft_Type		IM_Aircraft_Link
Add_Maint_Aircraft_Type		IM_Maint_Aircraft_Link
Remove_Maint_Aircraft_Type		IM_Maint_Aircraft_Link
First_Maint_Aircraft_Type		IM_Maint_Aircraft_Link
Next_Maint_Aircraft_Type		IM_Maint_Aircraft_Link
Add_Air_Component		IM_Air_Component_Link
Remove_Air_Component		IM_Air_Component_Link
First_Air_Component		IM_Air_Component_Link
Next_Air_Component		IM_Air_Component_Link

B.2.7 IM_Aircraft_Link. Links an airbase object to an aircraft object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of aircraft
Quantity	INTEGER	Number linked to airbase

Database Methods.

Method	Key?	Target Class
Create	No	Self
Get		Self
Put		Self
Delete		Self
Add_Airbase		IM_Airbase
Get_Airbase		IM_Airbase
Add_Aircraft_Type		IM_Aircraft

Method	Key?	Target Class
Get_Aircraft_Type		IM_Aircraft

B.2.8 IM_Aircraft_Package. Contains attributes needed to represent an aircraft package.

Attributes.

Attribute Name	Representation	Description
Mission_No	NATURAL	Unique mission id number
Force	Enumeration	Country of origin
HQ	STRING(1..5)	Headquarters of air package
Primary_Mission	Enumeration	Air Force mission type
Present_Location	INTEGER	Current hex location
Target_Id	NATURAL	Target identifier
Mission_Location	INTEGER	Hex location of target
Rqst_Prd_On_Target	INTEGER	Request period to bomb target
Rqst_Day_On_Target	INTEGER	Request day to bomb target
Loiter_Time	INTEGER	Time over target
Priority	INTEGER	Priority of target
Rendezvous_Hex	INTEGER	Hex aircraft will meet at
Region	INTEGER	Region to draw aircraft from
Distance	INTEGER	Target distance
Altitude	Enumeration	Cruise altitude
Speed	INTEGER	Package airspeed
Ineffective_Reason	Enumeration	
Orbit_Location	INTEGER	Orbit location for tankers
Detected	BOOLEAN	Has package been detected?
Positive_Id	BOOLEAN	Package identified?
Delayed	BOOLEAN	Has package been delayed a turn?
Was_Cancelled	BOOLEAN	Package cancelled?

Attribute Name	Representation	Description
Warhead	Enumeration	Type of weapon carried

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Target		(target objects)
Get_Target		(target objects)
Add_Primary_Sched		IM_Scheduled_Aircraft
Remove_Primary_Sched		IM_Scheduled_Aircraft
First_Primary_Sched		IM_Scheduled_Aircraft
Next_Primary_Sched		IM_Scheduled_Aircraft
Add_Escort_Sched		IM_Scheduled_Aircraft
Remove_Escort_Sched		IM_Scheduled_Aircraft
First_Escort_Sched		IM_Scheduled_Aircraft
Next_Escort_Sched		IM_Scheduled_Aircraft
Add_SEAD_Sched		IM_Scheduled_Aircraft
Remove_SEAD_Sched		IM_Scheduled_Aircraft
First_SEAD_Sched		IM_Scheduled_Aircraft
Next_SEAD_Sched		IM_Scheduled_Aircraft
Add_ECM_Sched		IM_Scheduled_Aircraft
Remove_ECM_Sched		IM_Scheduled_Aircraft
First_ECM_Sched		IM_Scheduled_Aircraft

Method	Key?	Target Class
Next_ECM_Sched		IM_Scheduled_Aircraft
Add_Refuel_Sched		IM_Scheduled_Aircraft
Remove_Refuel_Sched		IM_Scheduled_Aircraft
First_Refuel_Sched		IM_Scheduled_Aircraft
Next_Refuel_Sched		IM_Scheduled_Aircraft

B.2.9 IM_Aircraft. Contains attributes needed to represent an aircraft type.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of aircraft
Force	Enumeration	Country of origin
Common_Name	STRING(1..12)	Unofficial name
Night_CAP	FLOAT	
Weather_CAP	Enumeration	
Size	INTEGER	Size of aircraft
Avg_Sorties_Per_Week	INTEGER	Sortie capability
Search	INTEGER	Search capability
EC	INTEGER	
Max_Speed	INTEGER	Max speed of aircraft
Combat_Radius	INTEGER	Combat range of aircraft
Loiter_Time	INTEGER	Loiter time over area
Cargo	INTEGER	Cargo capacity
Recon_Ability	FLOAT	Recon. capability
Refuel	Enumeration	Mid-air refuelable?
Maint_Dist	Enumeration	
Maint_Mean	INTEGER	mean maintenance time
Maint_Stand_Dev	INTEGER	
Amt_Spares	INTEGER	

Attribute Name	Representation	Description
POL	INTEGER	Petrol, oil, lubr. reqmts.
Ramp	INTEGER	Ramp space required
Min_Runway_Needed	INTEGER	Minimum runway size
Air_Air_Rating	INTEGER	Air-to-Air combat ability
Air_Ground_Rating	INTEGER	Air-to-Ground combat ability
Max_Hex_Level	INTEGER	Max altitude
Weapons_Release_Level	INTEGER	Optimum weapons release alt.

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Target		(target objects)
Get_Target		(target objects)
Add_Conventional_Load		IM_Weapon_Load
Remove_Conventional_Load		IM_Weapon_Load
First_Conventional_Load		IM_Weapon_Load
Next_Conventional_Load		IM_Weapon_Load
Add_Biological_Load		IM_Weapon_Load
Remove_Biological_Load		IM_Weapon_Load
First_Biological_Load		IM_Weapon_Load
Next_Biological_Load		IM_Weapon_Load
Add_Nuclear_Load		IM_Weapon_Load

Method	Key?	Target Class
Remove_Nuclear_Load		IM_Weapon_Load
First_Nuclear_Load		IM_Weapon_Load
Next_Nuclear_Load		IM_Weapon_Load

B.2.10 IM_Base_Component. Represents an airbase component object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of component
Target_Weight	INTEGER	Weight of component
Length	INTEGER	Length of component
Width	INTEGER	Width of component

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Component_Link		IM_Air_Component
Remove_Component_Link		IM_Air_Component
First_Component_Link		IM_Air_Component
Next_Component_Link		IM_Air_Component

B.2.11 IM_Chemical_Weapon. Represents a chemical weapon object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of weapon
Force	Enumeration	Country of origin
Persistence	INTEGER	NBC effect time
Lethality	INTEGER	Quality of weapon
CEP	FLOAT	

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.12 IM_City. Represents a city object. Note that this object is used only by the user interface. The user interface represents locations as strings, so the attributes reflect this.

Attributes.

Attribute Name	Representation	Description
Name	STRING(1..12)	Name of city
Id	STRING(1..6)	Identification of city
Location	STIRNG(1..6)	Hex location
Size	NATURAL	Relative size of city
Capital	BOOLEAN	City is capital?

Attribute Name	Representation	Description
Population	NATURAL	Number of people in city

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.13 IM_Component_Link. Links a land unit object to a component object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of component
Quantity	INTEGER	Number linked to unit

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Unit		IM_Land_Unit

Method	Key?	Target Class
Get_Unit		IM_Land_Unit
Add_Component		IM_Ground_Component
Get_Component		IM_Ground_Component

B.2.14 IM_Depot. This object has the same attributes and methods as an airbase object.

B.2.15 IM_Forces. Contains attributes needed to represent a country.

Attributes.

Attribute Name	Representation	Description
Country	STRING(1..4)	Name of country
Force	Enumeration	Force id associated with country

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.16 IM_Ground_Component. Represents a ground component object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of component

Attribute Name	Representation	Description
Ammo_Usage_Rate	FLOAT	Rate of ammo use
POL_Usage_Rate	FLOAT	Rate of POL use
Hardware_Usage_Rate	FLOAT	Rate of hardware use
Target_Weight	INTEGER	Target priority of component
Firepower_Weight	INTEGER	Firepower ranking of component
Length	INTEGER	Length of component
Width	INTEGER	Width of component

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Component_Link		IM_Component_Link
Remove_Component_Link		IM_Component_Link
First_Component_Link		IM_Component_Link
Next_Component_Link		IM_Component_Link

B.2.17 IM_Ground_Hex. Contains attributes needed to represent an ground hex on the map board.

Attributes.

Attribute Name	Representation	Description
Lon	INTEGER	Longitude of Hex

Attribute Name	Representation	Description
Lat	INTEGER	Latitude of Hex
Hex_Weather	Enumeration	Current hex weather
Weather_Zone	INTEGER	Weather zone hex is in
Force	Enumeration	Country hex occupied by
Country	STRING(1..4)	Country hex located in
Center_Hex	NATURAL	Center of air hex grouping
Mission	Enumeration	ATK or DEF hex?
In_Contact	BOOLEAN	Adjacent hexes with enemy
In_Attrition	BOOLEAN	Adjacent hexes with enemy
Attrition	FLOAT	Rate of attrition in hex
CP_Out	FLOAT	Combat power leaving hex
CP_In	FLOAT	Combat power attacking hex
SAI	FLOAT	Aggregate SAI for units in hex
Persistence	INTEGER	Persistence of NBC weapons
EC	INTEGER	
Forest	INTEGER	Level of forestation
Terrain	INTEGER	Quality of terrain
Intel_Index	FLOAT	Intel quality for hex

Database Methods.

Method	Key?	Target Class	Notes
Create	Yes	Self	
Get_ObjID	Yes	Self	
First		Self	
Next		Self	
Get		Self	
Put		Self	
Delete		Self	

Method	Key?	Target Class	Notes
Add_XX_Pie_Piece		IM_Pie_Piece	Where XX=N,NE,E,SE,S,SW,W,
Get_XX_Pie_Piece		IM_Pie_Piece	Where XX=N,NE,E,SE,S,SW,W,
Add_XX_Hex_Side		IM_Hex_Side	Where XX=N,NE,E,SE,S,SW,W,
Get_XX_Hex_Side		IM_Hex_Side	Where XX=N,NE,E,SE,S,SW,W,
Add_Unit		IM_Land_Unit	
Remove_Unit		IM_Land_Unit	
First_Unit		IM_Land_Unit	
Next_Unit		IM_Land_Unit	
Add_Base		IM_Airbase	
Remove_Base		IM_Airbase	
First_Base		IM_Airbase	
Next_Base		IM_Airbase	
Add_Depot		IM_Depot	
Remove_Depot		IM_Depot	
First_Depot		IM_Depot	
Next_Depot		IM_Depot	

B.2.18 IM_Hardness. Linked to a target object, this object describes its hardness.

Attributes.

Attribute Name	Representation	Description
Target	STRING(1..15)	Target this applies to
Pk_Value	Enumeration	Hardness of this target

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self

Method	Key?	Target Class
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.19 IM_Hex_Side. Linked to one or two hexes, this describes the side attributes.

Attributes.

Attribute Name	Representation	Description
Hexside_No	NATURAL	Unique id of side
FEBA	BOOLEAN	FEBA hex?
Border	BOOLEAN	Side on border?
Coast	BOOLEAN	Side on coast?
River	Enumeration	Type of river on side

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.20 IM_Land_Mission. Contains attributes necessary to represent a land unit mission.

Attributes.

Attribute Name	Representation	Description
Order_Id	NATURAL	Unique id of mission
Target_Char	STRING(1..2)	First two chars of target id
Target_Id	NATURAL	Unique id of target
Day	INTEGER	Day of mission
Period	INTEGER	Period of mission
Mission	Enumeration	Type of mission

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Unit		IM_Land_Unit
Get_Unit		IM_Land_Unit
Add_Target		(target objects)
Get_Target		(target objects)

B.2.21 IM_Land_Unit. Contains attributes necessary to represent a land unit.

Attributes.

Attribute Name	Representation	Description
Land_Unit_Id	NATURAL	Unique unit id number
Type_Of_Unit	Enumeration	Army unit-type designation
Force	Enumeration	Country of origin
Present_Location	INTEGER	Current hex location
Current_Mission_No	INTEGER	Current mission
Current_Mission_Location	INTEGER	Location of target
Current_Army_Mission	Enumeration	Type of mission
Current_Mission	Enumeration	Priority of mission
Msn_Eff_Day	INTEGER	Day to start mission
Region	Enumeration	Unit's region
Hex_Dir	Enumeration	Direction of movement
Move_Allowed	Enumeration	Type of movement
In_Contact	BOOLEAN	In contact with enemy?
In_Attrition	BOOLEAN	In combat with enemy?
Attrition	FLOAT	Attrition suffered
Firepower	FLOAT	Inherent firepower of unit
Combatpower	FLOAT	Hex's combined power
Total_POL	INTEGER	POL Stocks
POL_Max_Capacity	INTEGER	Max POL stocks
POL_Usage_Rate	FLOAT	Rate of POL use
Total_Ammo	INTEGER	Ammo available
Ammo_Max_Capacity	INTEGER	Max ammo stocks
Ammo_Usage_Rate	FLOAT	Ammo use rate
Total_Hardware	INTEGER	
Hardware_Max_Capacity	INTEGER	Max hardware stocks
Hardware_Usage_Rate	FLOAT	Hardware use rate
Intel_Index	FLOAT	

Attribute Name	Representation	Description
Intel_Filter	FLOAT	Filter for reports
Breakpt	INTEGER	Point at which unit withdraws
Grid_Time	FLOAT	Time left until next move
Was_Intelled	BOOLEAN	Unit intelled by enemy?
Day_Last_Intelled	INTEGER	Day last intelled
Prd_Last_Intelled	INTEGER	Period last intelled
Loc_Last_Intelled	INTEGER	Location (hex) last intelled
Depot_Spt	NATURAL	Id of depot unit
Under_Chem_Nuc_Atk	BOOLEAN	Under NBC attack?
Mopp_Posture	INTEGER(1..4)	
Troop_Quality	INTEGER	Relative quality of troops
Groundspeed	INTEGER	Speed of unit
Fuel_Trucks	INTEGER	Number of fuel trucks
Ammo_Trucks	INTEGER	Number of ammo trucks
Water	INTEGER	Unit water supply stock
Water_Percent	FLOAT	
Water_Trucks	INTEGER	Number of water trucks
Engineers	INTEGER	Number of engineers
Eng_Vehicles	INTEGER	Engineers vehicles
Status	Enumeration	Overall status of unit
Unit_Size	INTEGER	
Corps_Id	NATURAL	Unique id of corps unit
Parent_Unit	NATURAL	Unique id of hq unit
Full_Designator	STRING(1..30)	Name of unit
Abbrev_Designator	STRING(1..8)	Short name of unit
Country	STRING(1..4)	Short name of country
Vis_To_Enemy	STRING(1..8)	Unit intel visibility

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Depot_Unit		IM_Land_Unit
Get_Depot_Unit		IM_Land_Unit
Add_Ground_Hex		IM_Hex
Get_Ground_Hex		IM_Hex
Add_Mission		IM_Land_Mission
Remove_Mission		IM_Land_Mission
First_Mission		IM_Land_Mission
Next_Mission		IM_Land_Mission
Add_Override_Mission		IM_Land_Mission
Remove_Override_Mission		IM_Land_Mission
First_Override_Mission		IM_Land_Mission
Next_Override_Mission		IM_Land_Mission
Add_Radar		IM_Radar
Remove_Radar		IM_Radar
First_Radar		IM_Radar
Next_Radar		IM_Radar
Add_Supported_Unit		IM_Land_Unit
Remove_Supported_Unit		IM_Land_Unit
First_Supported_Unit		IM_Land_Unit
Next_Supported_Unit		IM_Land_Unit

Method	Key?	Target Class
Add_Support_Unit		IM_Land_Unit
Remove_Support_Unit		IM_Land_Unit
First_Support_Unit		IM_Land_Unit
Next_Support_Unit		IM_Land_Unit
Add_Unit_Component		IM_Component_Link
Remove_Unit_Component		IM_Component_Link
First_Unit_Component		IM_Component_Link
Next_Unit_Component		IM_Component_Link
Add_Weapon		IM_Weapon_Link
Remove_Weapon		IM_Weapon_Link
First_Weapon		IM_Weapon_Link
Next_Weapon		IM_Weapon_Link
Add_Supply_Mission		IM_Supply_Mission
Remove_Supply_Mission		IM_Supply_Mission
First_Supply_Mission		IM_Supply_Mission
Next_Supply_Mission		IM_Supply_Mission

B.2.22 IM_Maint_Aircraft_Link. Links an airbase object to an aircraft object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of aircraft
Quantity	INTEGER	Number linked to airbase
Maint_Time_Remaining	INTEGER	Time in hours remaining

Database Methods.

Method	Key?	Target Class
Create	No	Self

Method	Key?	Target Class
Get		Self
Put		Self
Delete		Self
Add_Airbase		IM_Airbase
Get_Airbase		IM_Airbase
Add_Aircraft_Type		IM_Aircraft
Get_Aircraft_Type		IM_Aircraft

B.2.23 IM_Mission_Load. Holds id numbers of weapons loads for all combinations of weather, hardness, and mission.

Attributes.

Attribute Name	Representation	Description
Mission	Enumeration	Mission type
Load	INTEGER Array	Array of load id numbers

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
Get		Self
Put		Self
Delete		Self
Add_Weapons_Load		IM_Weapon_Load
Get_Weapons_Load		IM_Weapon_Load

B.2.24 IM_Nuclear_Weapon. Represents a nuclear weapon object.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of weapon
Yield	INTEGER	Yield in kilotons of TNT
Force	Enumeration	Country of origin
Persistence	INTEGER	NBC effect time
CEP	FLOAT	

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.25 IM_Obstacle. Represents an obstacle placed in a hex pie piece.

Attributes.

Attribute Name	Representation	Description
Obstacle_Id	NATURAL	Unique id number
Hexside_No	NATURAL	Hexside obstacle is on
Obstacle	Enumeration	Description of obstacle
Obs_Diff	Enumeration	Level of traversal difficulty
Vis_To_Enemy	STRING(1..8)	Visibility of obstacle

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.26 IM_Override_Mission. Same attributes and methods as the IM_Land_Mission object.

B.2.27 IM_Pie_Piece. Represents one-sixth of a hex and holds the trafficability of that segment.

Attributes.

Attribute Name	Representation	Description
Traffic_Type	Enumeration	Ease of trafficability of segment

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Obstacle		IM_Obstacle
Remove_Obstacle		IM_Obstacle

Method	Key?	Target Class
First_Obstacle		IM_Obstacle
Next_Obstacle		IM_Obstacle
Add_Pipeline_Segment		IM_Pipeline_Segment
Get_Pipeline_Segment		IM_Pipeline_Segment
Add_Railroad_Segment		IM_Railroad_Segment
Get_Railroad_Segment		IM_Railroad_Segment
Add_Road_Segment		IM_Road_Segment
Get_Road_Segment		IM_Road_Segment
Add_Parent_Hex		IM_Hex
Get_Parent_Hex		IM_Hex

B.2.28 IM_Pipeline_Segment. Contains the attributes for a segment of a pipeline.

Attributes.

Attribute Name	Representation	Description
Pipeline_Id	NATURAL	Unique segment number
Hex_No	NATURAL	Hex located in
Pie_Piece	Enumeration	Pie Piece located in
Product	Enumeration	Product carried
Name	STRING(1..12)	Name of pipeline
Flow	Enumeration	Flowing or not

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self

Method	Key?	Target Class
Get		Self
Put		Self
Delete		Self
Add_Pie_Piece		IM_Pie_Piece
Get_Pie_Piece		IM_Pie_Piece

B.2.29 IM_Radar. Contains the attributes for a unit radar object.

Attributes.

Attribute Name	Representation	Description
Type_Radar	Enumeration	Type of radar
Quality	FLOAT	Detection ability
Quantity	INTEGER	Number in use

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.30 IM_Railroad_Segment. Contains the attributes for a segment of a railroad.

Attributes.

Attribute Name	Representation	Description
Railroad_Id	NATURAL	Unique segment number
Hex_No	NATURAL	Hex located in

Attribute Name	Representation	Description
Pie_Piece	Enumeration	Pie Piece located in
Name	STRING(1..12)	Name of railroad
Flow	Enumeration	Flowing or not

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Pie_Piece		IM_Pie_Piece
Get_Pie_Piece		IM_Pie_Piece

B.2.31 IM_Road_Segment. Contains the attributes for a segment of a road.

Attributes.

Attribute Name	Representation	Description
Road_Id	NATURAL	Unique segment number
Hex_No	NATURAL	Hex located in
Pie_Piece	Enumeration	Pie Piece located in
Name	STRING(1..12)	Name of railroad
Size	Enumeration	Width/load ability of raod
Flow	Enumeration	Flowing or not

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Pie_Piece		IM_Pie_Piece
Get_Pie_Piece		IM_Pie_Piece

B.2.32 IM_Runway. Contains the attributes for a segment of a road.

Attributes.

Attribute Name	Representation	Description
Base_Id	NATURAL	Base attached to
Runway	INTEGER	
Condition	Enumeration	Condition of the runway
Current_Length	INTEGER	Current length in feet
Max_Length	INTEGER	Max length in feet

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self

Method	Key?	Target Class
Delete		Self
Add_Base		IM_Airbase
Get_Base		IM_Airbase

B.2.33 IM_SAM_Weapon. Contains the attributes for a surface-to-air weapon.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of weapon
Force	Enumeration	Country of origin
Warhead	Enumeration	Type of explosive
Slow_High	INTEGER	PK for slow,high targets
Slow_Low	INTEGER	PK for slow,low targets
Fast_High	INTEGER	PK for fast,high targets
Fast_Low	INTEGER	PK for fast,low targets
SSpk	FLOAT	
Miss_Radar_Range	Enumeration	Missile's radar range
Launcher_Rounds	INTEGER	Number of weapons
Reload_Time	INTEGER	Time between firings
Weather_Min	INTEGER	Required weather conditions

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self

Method	Key?	Target Class
Put		Self
Delete		Self

B.2.34 IM_Satellite. Represents a satellite object.

Attributes.

Attribute Name	Representation	Description
Satellite_Id	NATURAL	Unique satellite number
Designation	STRING(1..12)	Name of weapon
Force	Enumeration	Country of origin
Location	INTEGER	Air hex located in
Sat_Type	Enumeration	Type of satellite
Status	Enumeration	Current status of sat.
Speed	INTEGER	Speed of satellite
Direction	Enumeration	Direction of movement
Orbit	STRING(1..12)	Name of satellite orbit
Sat_Delay	INTEGER	Transmission delay time

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.35 IM_Scheduled_Aircraft. Links an aircraft package to its scheduled aircraft.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of aircraft
Quantity	INTEGER	Number linked to package

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Aircraft		IM_Aircraft
Get_Aircraft		IM_Aircraft

B.2.36 IM_SSM_Weapon. Contains the attributes for a surface-to-surface weapon.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of weapon
Force	Enumeration	Country of origin
Warhead	Enumeration	Type of explosive
Lethality_Radius	INTEGER	PK range
Cep	FLOAT	Quality of weapon
Pk_Hard_Point_Type	FLOAT	PK for hard target
Pk_Med_Point_Type	FLOAT	PK for med target
Pk_Soft_Point_Type	FLOAT	PK for soft target

Attribute Name	Representation	Description
Min_Range	INTEGER	Minimum firing arc
Max_Range	INTEGER	Max range of weapon
Launcher_Rounds	INTEGER	Number of weapons
Reload_Time	INTEGER	Time between firings

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.37 IM_Supply_Mission. Contains the attributes for a supply mission object.

Attributes.

Attribute Name	Representation	Description
Order_Id	INTEGER	Order mission is for
Type_Of_Supply	Enumeration	Type of supply carried
Delivery_Quantity	INTEGER	Amount delivered

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self

Method	Key?	Target Class
Next		Self
Get		Self
Put		Self
Delete		Self

B.2.38 IM_Supply_Train. Contains the attributes of a supply train object.

Attributes.

Attribute Name	Representation	Description
Supply_Train_Id	NATURAL	Unique supply id number
Resupply_Id	NATURAL	Resupply unit
Tot_Cap	INTEGER	Capacity of unit
In_Use	BOOLEAN	Being used?
Type_Of_ST	Enumeration	Type of supplies moved
Tran_Mode	Enumeration	How movement is done
Total_POL	INTEGER	Total POL carried
Total_Ammo	INTEGER	Total Ammo carried
Total_Hardware	INTEGER	Total hardware carried
Total_Spares	INTEGER	Total spares carried

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self

Method	Key?	Target Class
Add_Mission		IM_Supply_Mission
Remove_Mission		IM_Supply_Mission
First_Mission		IM_Supply_Mission
Next_Mission		IM_Supply_Mission
Add_Unit		IM_Land_Unit
Get_Unit		IM_Land_Unit
Add_Resupply		IM_Land_Unit
Get_Resupply		IM_Land_Unit

B.2.39 IM_Support_Link. Links a land unit to its supporting unit.

Attributes.

Attribute Name	Representation	Description
Unit_No	NATURAL	Unique id of support unit
Percent_Support	FLOAT	Percent of total support

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_To_Unit		IM_Land_Unit
Get_To_Unit		IM_Land_Unit
Add_From_Unit		IM_Land_Unit
Get_From_Unit		IM_Land_Unit

B.2.40 IM_Weapon_Link. Links a weapon to a land unit.

Attributes.

Attribute Name	Representation	Description
Designation	STRING(1..5)	Name of weapon
Weapons_Quantity	INTEGER	Amount
Launchers_Quanity	INTEGER	Number of launchers

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Unit		IM_Land_Unit
Get_Unit		IM_Land_Unit
Add_Weapon		(weapon objects)
Get_Weapon		(weapon objects)

B.2.41 IM_Weapon_Load. Object representing a load of a specific type of weapons.

Attributes.

Attribute Name	Representation	Description
Weapons_Load_Id	NATURAL	Unique no of this load
Designation	STRING(1..5)	Name of weapon
Quantity	INTEGER	Amount

Database Methods.

Method	Key?	Target Class
Create	No	Self
First		Self
Next		Self
Get		Self
Put		Self
Delete		Self
Add_Weapon		(weapon objects)
Get_Weapon		(weapon objects)

B.2.42 IM_Weather. Holds a weather forecast for a zone, on a specific day and period.

Attributes.

Attribute Name	Representation	Description
Zone	INTEGER	Zone of occurance
Day	INTEGER	Day of occurance
Period	INTEGER	Period of occurance
Good_Percent	INTEGER	Chances of good weather
Fair_Percent	INTEGER	Chances of fair weather
Weather	Enumeration	Actual weather

Database Methods.

Method	Key?	Target Class
Create	Yes	Self
Get_ObjID	Yes	Self
First		Self
Next		Self

Method	Key?	Target Class
Get		Self
Put		Self
Delete		Self

Vita

Captain Karl S. Mathias was born on 1 December, 1963 in Bartlesville, Oklahoma. He graduated from Snake River High School in Blackfoot, Idaho in 1982 and attended Utah State University majoring in computer science. While at USU, he was student commander of the AFROTC detachment, completing the program in 1986 as a Distinguished Graduate. He was then assigned to the 4702d Computer Services Squadron, Tyndall AFB, Florida as a Joint Surveillance System Programmer Analyst. He later became chief of the Force Management System project and directed its installation for 1st Air Force. In May, 1992, he entered the Air Force Institute of Technology's Graduate School of Engineering.

Upon completion of his studies at the Air Force Institute of Technology, Captain Mathias will be assigned to the Air Force Wargaming Center at Maxwell AFB, Alabama.

Permanent address: 75 South 601 West
Blackfoot, Idaho 83221

Bibliography

1. Associates, O'Reilly &. *X Window System in a Nutshell.* O'Reilly & Associates, Inc., 1992.
2. Atkinson, Malcolm and others. "An Approach to Persistent Programming," *Readings in Object-Oriented Database Systems* (1990).
3. Atkinson, Malcolm and others. "The Object-Oriented Database System Manifesto," *Deductive and Object-oriented Databases* (1990).
4. Barry, Douglas K. "OODBMS Feature Listing," *Object Magazine* (January-February, 1993).
5. Booch, Grady. *Software Components with Ada: Structures, Tools, and Subsystems.* Benjamin/Cummings Publishing Company, Inc., 1987.
6. Douglass, David Scott. *Object-Oriented Analysis, Design, and Implementation of the Saber Wargame.* MS thesis, AFIT/GCS/ENG/92D-02, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1992.
7. Freese, Tim. "Ada/X Interface: Binding versus Implementation." *Proceedings of the Tri-Ada 1992 Conference.* 478. 1992.
8. Heller, Dan. *Motif Programming Manual.* O'Reilly & Associates, Inc., 1991.
9. Horton, Andrew. *Design and Implementation of a Graphical User Interface and Database Management system for the Saber Wargame.* MS thesis, AFIT/GCS/ENG/91D-08, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1991.
10. Klabunde, Gary Wayne. *An Animated Graphical Postprocessor for the Saber Wargame.* MS thesis, AFIT/GCS/ENG/91D-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1991.
11. Lewin, Stu. "And the Winner Is ...." *Proceedings of the Tri-Ada 1992 Conference.* 479. 1992.
12. Mann, William F. III. *Saber, A Theater Level Wargame.* MS thesis, AFIT/GOR/ENS/91M-09, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, March 1991.
13. McKay, Amber M. "A Layered Architecture for DBMS Interactions with an Ada Application." *9th Annual National Conference on Ada Technology.* 152–159. 1991.
14. Moore, Donald Ray. *An Enhanced User Interface for the Saber Wargame.* MS thesis, AFIT/GCS/ENG/92D-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1991.
15. Ness, Marlin Allen. *A New Land Battle for the Theater War Exercise.* MS thesis, AFIT/GE/ENG/90J-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, January 1990.
16. Nye, Adrian. *Xlib Programming Manual.* O'Reilly & Associates, Inc., 1990.

17. Nye, Adrian and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual*. O'Reilly & Associates, Inc., 1990.

18. Open Systems Foundation. *OSF/Motif Style Guide*. Prentice Hall, 1990.

19. Perla, Peter P. *The Art of Wargaming*. Naval Institute Press, 1990.

20. Rumbaugh, James and Others. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

21. Sherry, Christine M. *Object-Oriented Analysis and Design of the Saber Wargame*. MS thesis, AFIT/GCS/ENG/91D-21, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1991.

22. Software Productivity Solutions, Inc. *Classic-Ada User's Manual*, 1989.

23. Sun Microsystems, Inc. *SunAda Reference Manual*, 1992.

24. Systems Engineering Research Corporation. *Ada/ Motif User's Manual*, 1992.

25. The Committee for Advanced DBMS Function. *Third-Generation Data Base System Manifesto*. Technical Report Memorandum No. UCB/ERL M90/28, University of California, Berkely, 1990.

26. Voketaitis, Arnold M. "A Portable and Reusable RDBMS Interface for Ada," *ACM Ada Letters*, 64–76 (September 1992).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
INTEGRATION AND ENHANCEMENT OF THE SABER WARGAME

5. FUNDING NUMBERS

6. AUTHOR(S)
Karl S. Mathias, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-15

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AU CADRE/WG
Maxwell AFB, AL 36112

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Saber wargame is a theater-level air/land battle wargame written in Ada that is being developed for the Air Force Wargaming Center at Maxwell AFB, AL. This thesis documents how the user interface and simulation engine were integrated. Integration was accomplished by developing a portable object-oriented database system (OODBMS) interface. The interface was implemented in Ada and tied to an OODBMS also written in Ada. Using the interface, both subsystems were able to work from a consistent database and exchange information. The user interface was enhanced by converting it from the Software Technology for Adaptable Reliable Systems Ada/X Window System bindings to a newer commercial set. Generic components were constructed to allow the rapid development of Motif input forms written in Ada.

14. SUBJECT TERMS
wargame simulation, software engineering, Ada,
databases, object-oriented databases, graphical user interfaces
bindings, X Window System

15. NUMBER OF PAGES
159

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL